A library for parallel arithmetic using a modular representation

Power, David James

Award date:
2001

Awarding institution:
University of Bath

A Library for Parallel Arithmetic using a Modular Representation

submitted by

David James Power

for the degree of Doctor of Philosophy

of the

University of Bath

2001

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Acknowledgements

I would like to thank my supervisor Russell Bradford for his help, advice and lending me his house. I would also like to thank the people at JICS at the University of Tennessee for the use of their Maspar MP-2. I would like to thank Julian Padget for looking after me when Russell was away. I would like to thank all the people of 1W2.26 for keeping me distracted from writing my thesis, but most importantly of all, I would like to thank my family for their love and support.

Summary

This thesis is concerned with performing integer calculations using numbers of the order of thousands of decimal digits on a parallel computer. The traditional representation of numbers is contrasted with a modular representation where numbers are stored as a list of residues to co-prime moduli.

By using an approximate Chinese Remainder Theorem reconstruction it is shown how it is possible to perform all the basic arithmetic operations without leaving a modular representation, in particular, efficient parallel solutions are demonstrated for both comparison and general division.

A library of functions are described which are based around a datatype consisting of a set of residues coupled with an approximate reconstruction of the number. This library hides both the datatype and the parallelism from the end user allowing standard sequential algorithms to be directly coded.

A range of applications are demonstrated including GCD, division by a fixed divisor and determinant calculations. Results are shown using a range of hardware including clusters of workstations, shared memory machines and a 4096 processor SIMD machine.

Contents

List of Tables

List of Figures

List of Algorithms

Glossary of Symbols

$\boldsymbol{Crt(X)}$ The result of a Chinese Remainder Theorem reconstruction of X. See subsection 2.2.1.

$\boldsymbol{d}$ The bits of precision used in approximate Chinese Remainder Theorem reconstruction. See subsection 2.2.3.

$\boldsymbol{d_i}$ The i^{th} digit of a Mixed Radix Conversion. See subsection 2.3.2.

$\boldsymbol{E}$ The output of the function `approxCRT`. See subsection 3.6.1.

$\boldsymbol{\hat{E}}$ The output of the function `t_approxCRT`. See subsection 3.6.3.

$\boldsymbol{E_i}$ Estimate of i^{th} term in an approximate Chinese Remainder Theorem reconstruction. See subsection 2.2.3.

$\boldsymbol{e_i}$ Error in estimate of i^{th} term in an approximate Chinese Remainder Theorem reconstruction. See subsection 2.2.3.

$\boldsymbol{Inv_i}$ Constant needed in a Chinese Remainder Theorem reconstruction. See subsection 2.2.1.

$\boldsymbol{iproc}$ The processor index. See subsection 1.2.1.

$\boldsymbol{k}$ This is the integer part of an Approximate Chinese Remainder Theorem reconstruction. See subsection 2.2.2.

$\boldsymbol{len(X)}$ The length of integer X when stored in a modular representation. See subsection 2.3.1.

$\boldsymbol{M}$ The product of all moduli. See subsection 2.2.1.

$\boldsymbol{M(n)}$ The product of the first n moduli. See subsection 3.7.2.

$\boldsymbol{M_i}$ The result of dividing M by the i^{th} modulus. See subsection 2.2.1.

$\boldsymbol{m_i}$ Error in estimate of i^{th} term in an approximate Chinese Remainder Theorem reconstruction expressed as an integer. See subsection 2.2.3.

$\boldsymbol{n}$ The number of moduli. See subsection 2.2.1.

$\boldsymbol{nproc}$ The number of processors. See subsection 1.2.1.

$\boldsymbol{p_i}$ The i^{th} modulus. See subsection 2.2.1.

$\boldsymbol{X}$ The integer which is stored in the modular representation. See subsection 2.2.1.

$\boldsymbol{x}$ The number of extra moduli required in a base extension. See subsection 2.4.1.

$\boldsymbol{x_i}$ The i^{th} residue of the number X. See subsection 2.2.1.

$\boldsymbol{y_i}$ The product of x_i and Inv_i modulo p_i. A midway point in a Chinese Remainder Theorem reconstruction. See subsection 2.2.1.

$\boldsymbol{\delta}$ The number of extra moduli need to store the result of a multiplication. See subsection 3.11.1.

Part I

Parallel Modular Arithmetic

Chapter 1

Introduction

1.1 Bignum arithmetic

1.1.1 What is a Bignum?

A microprocessor is capable of performing calculations on a restricted range of integers. If you wish to perform calculations on integers which do not lie in this range then the calculations need to be performed in software. These integers, which are too large for the microprocessor, are often called *bignums*.

Section 1.1.1 contains an overview of how arithmetic can be performed on these bignums using a traditional representation on a sequential computer. As this thesis is concerned with performing these calculations in parallel this is discussed in Section 1.2. Examples are drawn from both parallel computing and circuit design as much research in parallel arithmetic is aimed at building custom circuits to perform specific calculations such as RSA[34] decoding.

In Section 1.3, an alternative modular representation is introduced which is more suited to parallel computation. Previous work on solving problems using this representation is discussed in this section along with how it relates to the main body of work being presented.

Finally, in Section 1.4 the structure of the thesis is discussed, along with a description of its main aims and objectives.

1.1.2 Traditional representations

The traditional method of representing a bignum in a computer is as a sequence of integers which are less than the processor integer limit. Each member of the sequence is a digit of the bignum written to a certain number base. The base chosen will depend on the implementation. If speed of input/output is important, a power of 10 may be used; if speed of calculation is important, a power of 2 may be used. To represent a negative number either the most significant digit may be signed, or the sign of the number may be stored separately. Unlike paper and pencil arithmetic the least significant digit is usually stored first.

In the examples that follow it will be assumed that the number base is B, and that the sign is stored separately. If the number in question is X, the i^{th} digit will be x_i, $lenX$ will be the number of significant digits, and $signX$ the sign. This is all summarised below.

$$|X| = \langle x_0, x_1, \ldots, x_{n-1} \rangle = x_0 + x_1 B + \ldots + x_{n-1} B^{n-1} \tag{1.1}$$

$$lenX = MAX\{i | x_i \neq 0\} + 1 \tag{1.2}$$

$$signX = \begin{cases} 0 & \text{if } X = 0 \\ X/|X| & \text{otherwise} \end{cases} \tag{1.3}$$

1.1.3 Traditional algorithms

In *The Art of Computer Programming* Volume 2 [26] there are descriptions of the *Classical Algorithms* for bignum arithmetic. These deviate only slightly from the paper-and-pencil algorithms taught at school. Below is a brief summary of these algorithms by which all other algorithms can be compared. Note that only positive numbers are considered, as signed numbers will add unnecessary complication, without affecting the core workings of any of the following algorithms. As a slight deviation from the original, the lengths of the results will also be calculated.

Addition

Given two positive numbers X and Y, with $lenX = n$, $lenY \leq n$, we wish to calculate $Z = X+Y$ and its length $lenZ$. This can be achieved using Algorithm 1.

Throughout the following algorithms $(digit, carry) \leftarrow DivRem(A, B)$, represents the assignment of $digit \leftarrow A \bmod B$, and $carry \leftarrow \lfloor A/B \rfloor$.

Looking at the algorithm it is clear that the time complexity of addition is $O(n)$, and that the dependence on the carry will hinder parallelisation.

Algorithm 1 Classical Addition of two bignums

```
1: carry ← 0
2: for i = 0 to n − 1 do
3:    (z_i, carry) ← DivRem(x_i + y_i + carry, B)
4: end for
5: z_n ← carry
6: if z_n ≠ 0 then
7:    lenZ ← n + 1
8: else
9:    lenZ ← n
10: end if
```

Subtraction

Given two positive numbers X and Y, $X \geq Y$, $lenX = n$, we wish to calculate $Z = X - Y$, and its length $lenZ$. This can be achieved using Algorithm 2. The algorithm is similar to addition sharing its time complexity of $O(n)$, and the dependence on a carry. Note that the carry is either 0 or -1.

Algorithm 2 Classical Subtraction of two bignums

```
1: carry ← 0
2: for i = 0 to n − 1 do
3:    (z_i, carry) ← DivRem(x_i − y_i + carry, B)
4:    if z_i ≠ 0 then
5:       lenZ ← i + 1
6:    end if
7: end for
```

Comparison

In Algorithm 2 it is assumed that $X \geq Y$ so that the result of the subtraction is positive. To ensure that this is the case a comparison algorithm is needed. Algorithm 3 takes as input two positive integers X and Y, $lenX, lenY \leq n$ and

returns, -1 if $X < Y$, 0 if $X = Y$ and 1 if $X > Y$. In the best case the time complexity of this algorithm is $O(1)$, but in the worst case it is $O(n)$.

Algorithm 3 Classical Comparison of two bignums

```
1: if lenX ≠ lenY then
2:   if lenX > lenY then
3:     return 1
4:   else
5:     return -1
6:   end if
7: end if
8: for i = lenX − 1 to 0 do
9:   if x_i ≠ y_i then
10:    if x_i > y_i then
11:      return 1
12:    else
13:      return -1
14:    end if
15:  end if
16: end for
17: return 0
```

Multiplication

Given two positive numbers X and Y, $lenX = n, lenY = m$, we wish to calculate $Z = X*Y$, and its length $lenZ$. This can be achieved using Algorithm 4. Looking at the algorithm it is clear that the time complexity of multiplication is $O(nm)$.

Division

Given two positive integers X and Y, $lenX = m + n$, $lenY = n$, we wish to calculate Q and R such that $X = QY + R$, $0 \leq R < Y$. The length of Q will be at most $m + 1$ and the length of R will be at most n. This calculation can be performed using Algorithm 5.

This algorithm is significantly more complicated then the others and consists of three main stages. In the first stage, the numbers are scaled up by a factor d so that highest digit of the divisor is larger than half the number base, this is the normalisation step. In the second stage, each digit of Q is calculated in turn

Algorithm 4 Classical Multiplication of two bignums

1: **for** $i = 0$ **to** $m - 1$ **do**
2: $z_i \leftarrow 0$;
3: **end for**
4: **for** $i = 0$ **to** $n - 1$ **do**
5: $carry \leftarrow 0$
6: **for** $j = 0$ **to** $m - 1$ **do**
7: $(z_{i+j}, carry) \leftarrow DivRem(x_i y_j + z_{i+j} + carry, B)$
8: **end for**
9: $z_{i+m} \leftarrow carry$
10: **end for**
11: **if** $z_{n+m-1} \neq 0$ **then**
12: $lenZ \leftarrow n + m$
13: **else**
14: $lenZ \leftarrow n + m - 1$
15: **end if**

from the highest to the lowest. In the last stage, the remainder is divided by d, which is the un-normalisation step.

The majority of the time is spent in the second stage, where in each step an estimate of the quotient digit $\hat{q}$ is made. This estimate is certain to be either correct or one too large, see [26]. The remaining value of X is then reduced by $\hat{q}YB^i$. If this result is negative then $q_i = \hat{q} - 1$, and YB^i is added back onto X.

Despite being significantly more complicated than the algorithm for multiplication the time complexity is the same, $O(mn)$.

1.1.4 Improvements in traditional algorithms

Karatsuba Multiplication

One method of speeding up multiplication was proposed by A. Karatsuba in 1962. Given two numbers X and Y both of length $2n$. Then Equation 1.4 holds, where $0 \leq X_0, X_1, Y_0, Y_1 < B^n$.

$$XY = (X_1 B^n + X_0)(Y_1 B^n + Y_0) \tag{1.4}$$

$$XY = (B^{2n} + B^n) X_1 Y_1 + B^n (X_1 - X_0)(Y_0 - Y_1) + (B^n + 1) X_0 Y_0$$

This reduces the multiplication of two $2n$ digit numbers to the multiplication

Algorithm 5 Classical Division of two bignums

1: $d \leftarrow \lfloor (B-1)/y_{n-1} \rfloor$
2: $carry \leftarrow 0$
3: **for** $i = 0$ **to** $m+n$ **do**
4: $\quad (x_i, carry) \leftarrow DivRem(x_i d + carry, B)$
5: **end for**
6: $carry \leftarrow 0$
7: **for** $i = 0$ **to** $n-1$ **do**
8: $\quad (y_i, carry) \leftarrow DivRem(y_i d + carry, B)$
9: **end for**
10: $y_n \leftarrow 0$
11: **for** $i = m$ **to** 0 **do**
12: $\quad (\hat{r}, \hat{q}) \leftarrow DivRem(x_{i+n}B + x_{i+n-1}, y_{n-1})$
13: $\quad$ **while** $(\hat{q} = B$ **or** $\hat{q}y_{n-2} > B\hat{r} + x_{i+n-2})$ **do**
14: $\quad\quad \hat{q} \leftarrow \hat{q} - 1$
15: $\quad\quad \hat{r} \leftarrow \hat{r} + y_{n-1}$
16: $\quad$ **end while**
17: $\quad carry \leftarrow 0$
18: $\quad$ **for** $j = 0$ **to** n **do**
19: $\quad\quad (x_{i+j}, carry) \leftarrow DivRem(x_{i+j} - \hat{q}y_j + carry, B)$
20: $\quad$ **end for**
21: $\quad q_i \leftarrow \hat{q}$
22: $\quad$ **if** $carry \neq 0$ **then**
23: $\quad\quad carry \leftarrow 0$
24: $\quad\quad$ **for** $j = 0$ **to** n **do**
25: $\quad\quad\quad (x_{i+j}, carry) \leftarrow DivRem(x_{i+j} + y_j + carry, B)$
26: $\quad\quad$ **end for**
27: $\quad\quad q_i \leftarrow q_i - 1$
28: $\quad$ **end if**
29: **end for**
30: $lenQ \leftarrow m - 1$
31: **if** $q_m \neq 0$ **then**
32: $\quad lenQ \leftarrow m$
33: **end if**
34: $carry \leftarrow 0$
35: $lenR \leftarrow 0$
36: **for** $i = n$ **to** 0 **do**
37: $\quad (carry, r_i) \leftarrow DivRem(x_i + carryB, d)$
38: $\quad$ **if** $lenR = 0$ **and** $x_i \neq 0$ **then**
39: $\quad\quad lenR \leftarrow i$
40: $\quad$ **end if**
41: **end for**
42: $lenR \leftarrow lenR + 1$

of three n bit numbers plus some subtractions and digit shifting all of which will take time of $O(n)$. The method can be applied recursively, giving the following relationship for the time complexity of Karatsuba multiplication.

$$T(2n) \leq 3T(n) + O(n) \tag{1.5}$$

This can be shown to reduce to $T(n) = O(n^{\log_2 3})$: see [26] for details. The method can be generalised into a method involving breaking the number into more and more pieces, which can lead to a complexity of $O(n^{1+\epsilon})$ for any chosen ϵ, though it is rarely done as the constant factors grow rapidly.

This type of multiplication is used in most bignum packages as the constant factors involved are relatively small. Typical number lengths (in machine words) which are needed to obtain speedup over classical multiplication, are between 8 and 32 depending on the efficiency of the two implementations.

FFT Multiplication

In terms of time complexity, one of the fastest algorithms for multiplying numbers stored in a traditional representation is Fast Fourier Transform (FFT) based multiplication.

The Fast Fourier Transform is an algorithm for performing Discrete Fourier Transforms (DFTs), which are Fourier Transforms of discrete sequences.

The Discrete Fourier transform of the sequence $\langle x_0, x_1, \ldots, x_{m-1} \rangle$ is the sequence $\langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{m-1} \rangle$, where the value of $\hat{x}_j$ is defined by Equation 1.6 in which $\omega = e^{2\pi i/m}$ is an m^{th} root of unity.

$$\hat{x}_j = \sum_{k=0}^{m-1} \omega^{jk} x_j. \tag{1.6}$$

The reason that a Discrete Fourier Transform is useful in multiplying numbers is related to Equation 1.7. Note that X and Y are assumed to be of length n or less.

$$XY = \sum_{k=0}^{2n-2} \left(\sum_{i+j=k} x_i y_j \right) B^k \tag{1.7}$$

Starting with two sequences $\langle x_0, x_1, \ldots, x_{m-1} \rangle$ and $\langle y_0, y_1, \ldots, y_{m-1} \rangle$, after

a Fourier Transform they will become $\langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{m-1} \rangle$ and $\langle \hat{y}_0, \hat{y}_1, \ldots, \hat{y}_{m-1} \rangle$ respectively. A third sequence can then be formed by multiplying matching pairs together to form $\langle \hat{x}_0\hat{y}_0, \hat{x}_1\hat{y}_1, \ldots, \hat{x}_{m-1}\hat{y}_{m-1} \rangle$. The inverse Fourier Transform of this third sequence will be $\langle z_0, z_1, \ldots, z_{m-1} \rangle$ where the individual members obey Equation 1.8.

$$z_k = \sum_{i+j \equiv k \bmod m} x_i y_j \tag{1.8}$$

In order to use this method to multiply two bignums, the value of m can be set to $2n$, this will result in an extra n values for each of the sequences representing X and Y. These extra values are set to zero, resulting in Equation 1.7 becoming:

$$XY = \sum_{k=0}^{2n-2} z_k B^k. \tag{1.9}$$

Although it appears that Equation 1.9 is the correct form for a traditional representation, each of the z_k is likely to be greater than the number base. For this reason a final carry phase is required, where starting from the least significant digit a carry is propagated up the sequence of digits.

If m is set to be a power of 2 a Fast Fourier Transform can be used, using this the time taken will be $T(n) = O(nT(\log n)) = O(n \log n \log\log n \ldots)$, see [26] for details. Although this sounds very attractive, speedup over the Karatsuba algorithm is not achieved until the numbers become very large. One study [5] found that the cross over was between 1024 and 2048 bytes. This should be compared with a cross over of just 32 bytes for Karatsuba over Classical multiplication.

Division using a Newtonian Inverse

An estimate of the value of $1/Y$ can be made using a Newtonian iteration.

$$z_{i+1} = 2z_i - Yz_i^2 \tag{1.10}$$

This will quickly converge on the true value of $1/Y$ as long as the initial estimate z_0 is accurate enough. If we assume that $z_i = (1 - \epsilon)/Y$ then $z_{i+1} = (1 - \epsilon^2)/Y$, which is a doubling of precision.

If $Y > 1$ the value $1/Y$ will not be a integer. To be able to calculate the

inverse, the estimate will need to be stored as Z where $Z/B^m \approx 1/Y$. At each step of the calculation the size of Z will double requiring multiplication of ever larger integers.

Using the algorithm described in [26] the time to calculate an n digit inverse is $T(n) = 2M(4n) + 2M(2n) + 2M(n) + 2M(n/2) + \ldots + O(n)$. Where $M(n)$ is the time taken to perform an n digit multiplication. If $M(n) = \Omega(n)$, then $T(n) = O(M(n))$.

Once the inverse has been calculated, the value of $q = \lfloor X/Y \rfloor$ can be approximated as $\hat{q} = \lfloor ZX/B^n \rfloor$. Assuming that $len(X) \leq n$, an n digit inverse will be sufficient for $\hat{q}$ to be within 1 of the correct answer. This can be checked by calculating $\hat{r} = X - \hat{q}Y$, if $0 \leq \hat{r} < Y$, then $q = \hat{q}$ and $r = \hat{r}$ otherwise Y can be added or subtracted as necessary.

The time complexity of division using this method is the same as the time complexity of multiplication (assuming $M(n) = \Omega(n)$). By using a fast multiplication algorithm it should be possible to divide much quicker than using the Classical method. However, like the FFT multiplication this method is not practical for small numbers. Indeed one study [5], showed that using either a Karatsuba or an FFT based multiplication a divisor of 4096 bytes was needed to achieve speedup.

Jebelean Exact Division

In [24] a method is proposed that allows the result of a division to be built up from the least significant bits first. If it is assumed that $Z = X * Y$ and that Z and X are known, then if the division is exact the least significant bits of Y will be equal to $Z * X^{-1} \bmod B$ where B is the number base. Note, that if X is not relatively prime to B, bit shifting can be carried out prior to the calculation.

Once the least significant bits of Y are known then it is possible to reduce the size of Z by subtracting off X multiplied by the least significant bits of Y. This will result in a new Z, in which the least significant bits are zero. Z can then be bit shifted by the size of the number base and the whole process repeated.

The process can be further speeded up by limiting the precision to which Z is recalculated. Using this method a speedup of approximately 2 times can be achieved over the traditional method of division.

The paper also describes a method of performing a GCD calculation us-

ing this exact division method, which was found to be approximately 10 percent quicker than a modified Euclidean GCD calculation, where $gcd(A, B) \rightarrow gcd(B, min(A \bmod B, B - (A \bmod B)))$.

Montgomery reduction

One very practical way to speed modular multiplication was proposed in [30].

The basic principle is this, given a modulus N that we need to work with, and another modulus R in which arithmetic is simple (e.g. 2^n), then it is possible to calculate the value $TR^{-1} \bmod N$ from T using only calculations mod$R$. The only constraints are that $R > N$, $gcd(R, N) = 1$ and $0 \leq T < RN$.

To perform the calculation two integers R^{-1} and N' are needed which must obey the following constraints, $0 < R^{-1} < N$, $0 < N' < R$ and $RR^{-1} - NN' = 1$. These can be calculated using the extended Euclidean algorithm in the standard way. The reduction then proceeds in three steps.

- m $\leftarrow (T \bmod R)N' \bmod R$ [so $0 \leq m < R$]
- $t \leftarrow (T + mN)/R$
- if $t \geq N$ then return $t - N$ else return t

To use this reduction to perform modular multiplications numbers are first converted to a Montgomery representation. The Montgomery representation of the integer X is $XR \bmod N$. This can be multiplied by the Montgomery representation of Y to form $T = (XR \bmod N)(YR \bmod N)$. As $T < N^2 < NR$ a Montgomery reduction can be performed to produce $TR^{-1} \bmod N = (XY)R \bmod N$ which is the Montgomery representation of XY.

To convert a number into a Montgomery representation the number can be multiplied by $R^2 \bmod N$ and then a Montgomery reduction can be performed. A subsequent Montgomery reduction will reverse the conversion returning the original number.

1.1.5 Common Packages

Many packages are capable of performing calculations on bignums. These include languages such as LISP, computer algebra systems such as Reduce, and programming libraries such as GMP [16]. In this thesis it is planned to build a library of

functions in the style of GMP, that is a set of datatypes with associated functions to perform all basic and some higher order arithmetic tasks.

Unlike other systems it is planned to perform the arithmetic operations on a range of parallel computers. This presents a new set of challenges which are discussed in Section 1.2.

1.2 Parallel arithmetic

1.2.1 Terminology used in parallel computing

Parallel computing is concerned with using multiple processing units to solve a single problem. By using multiple processing units it may be possible to solve problems faster or to cope with larger problem sizes.

In this section some of the terminology used in parallel computing will be described.

Parallel Architectures

The are many ways of classifying parallel computers, one commonly used system is was proposed by M.J. Flynn in [15]. Flynn's classification is based on instruction and data streams.

- SISD (Single Instruction stream, Single Data stream)
- SIMD (Single Instruction stream, Multiple Data streams)
- MIMD (Multiple Instruction streams, Multiple Data streams)

There is also MISD (Multiple Instruction streams, Single Data stream) but that is not normally used.

A standard sequential computer would be classified as SISD.

In a SIMD computer each processor must execute exactly the same instruction at the same time. If there is a branch in the code different processors may need to execute different blocks of code. In this case one set of processors will wait while the other set execute their instructions.

In a MIMD computer there are no such restrictions, this is the most flexible type of architecture. Each processor has its own set of instructions.

Synchronous/Asynchronous

A parallel computer is synchronous if all the processors act in step with one another. This is the case for SIMD computers.

A parallel computer is asynchronous if the processors act out of step with each other. A MIMD computer is usually asynchronous, but this does not have to be the case.

While asynchronous computers give the maximum flexibility, it is possible for the processors to get too far out of step. For example, one processor may try to use data that another processor hasn't finished calculating. This can give unpredictable results.

One method of ensuring each of the computers has reached a certain critical point in the calculation is to have a synchronisation barrier. At a synchronisation barrier each processor waits until it is sure the other processors have also reached the barrier.

Code which is written with many synchronisation barriers is sometimes referred to as semi-synchronous.

Distributed/shared memory

In a distributed memory parallel computer each processor has its own private memory store. In a shared memory computer all the processors use the same memory store.

Shared memory gives many advantages in terms of speed and ease of communication. However it is difficult to build shared memory machines with large numbers of processors.

Distributed memory machines require separate communication hardware to enable the processors to communicate. A cluster of workstations can be thought of as a distributed memory parallel computer.

The PRAM model

The PRAM (Parallel Random Access Machine) model is a theoretical model of a parallel computer. A PRAM consists of *nproc* processors all of which have access to a memory of unbounded size. The processors share a common clock but may execute different instructions in each clock cycle.

Using the classifications we have built up previously it is a synchronous shared memory MIMD computer.

There are four variants of this model which differ in the ability of more than one processor to read or write a memory location concurrently.

- EREW (Exclusive Read Exclusive Write)
  Processors may neither read nor write to the same memory location concurrently
- CREW (Concurrent Read Exclusive Write)
  Processors may read but not write to the same memory location concurrently
- ERCW (Exclusive Read Concurrent Write)
  Processors may write to but not read the same memory location concurrently
- CRCW (Concurrent Read Concurrent Write)
  Processors may both read and write to the same memory location concurrently

Concurrent reading of the same memory location does not cause any semantic problems, however concurrent writing can be handled in several different ways. Methods for dealing with problem include the following.

- Common: All processors must write the same value, or they will all fail
- Arbitrary: An arbitrary processor will succeed in writing, the rest will fail
- Priority: Each processor has its own unique priority. The processor with the highest priority will succeed
- Combine: All the values written are combined according to some rule, such as summation
- Undefined: Writes succeed, but with an undefined result

Parallel Efficiency

If the time taken to perform the calculation on one processor is t_1, and the time taken to perform the same calculation on *nproc* processors is t_{nproc}, then the speedup S_{nproc} is defined as

$$S_{nproc} = \frac{t_1}{t_{nproc}}. \tag{1.11}$$

The parallel efficiency of a calculation E_{nproc} is the speedup divided by the number of processors.

$$E_{nproc} = \frac{S_{nproc}}{nproc} \tag{1.12}$$

If the efficiency of a parallel computation is greater than 1, then the speedup must be greater than the number of processors. This is known as super-linear speedup.

Super-linear speedup is usually due to the increased resources of the parallel computer. In particular the total amount of processor cache will increase in proportion to the number of processors.

Processor Index

It is usual to give each processor its own unique index. This will usually lie in the range 0 to $nproc - 1$. This will be referred to as *iproc* throughout this thesis.

Parallel Reduction

In a parallel reduction, values from each of the processors are combined together to yield a single value. A common example of this would be a parallel summation where the values are combined using addition.

Others operators commonly used are multiplication, AND, OR, MIN and MAX. All these operations are associative when integers are used, allowing calculations to be performed in parallel, they are also commutative which gives more freedom in implementation, though it is not necessary.

The minimum number of steps required to perform a parallel reduction is $\lceil \log_2(nproc) \rceil$, this can be achieved using a binary tree.

Each processor starts with an index value which is initially set to *iproc*. In each stage of the reduction the processor with index $2k + 1$ will pass its value to the processor with index $2k$. In the next stage, all the processors with an odd index will drop out of the calculation and the remaining processors will divide their index by two. The calculation will end when there is only one processor active.

If each of the processors is connected to all others, then each step will take a constant amount of time. This will lead to a time complexity of $O(\log(nproc))$.

Broadcasting and globalor

A broadcast is when a value stored on a single processor is passed to all of the other processors. If we consider a CREW PRAM, a broadcast can be performed by one processor writing to a memory location in one step and all the processors reading that value in the next step. This takes a constant amount of time.

A globalor operation is another name for a reduction using logical OR. It is called globalor to differentiate it from the other reduction operations as it can often be performed in a different way.

Using a CRCW PRAM with arbitrary concurrent writing, a globalor can be performed in three steps. In the first step, the value FALSE is written to a memory location. In the second step, all the processors which have a TRUE value write the value TRUE to the same memory location. In the last step, all the processors read the value which will be the globalor.

To use AND instead of OR the values written should be logically inverted, as should the result, this is an instance of de Morgan's Law.

In a real world computer, similar effects can be achieved by wiring all of the processors to a single point. It will then be possible to send data to all processors simultaneously from that point. A globalor operation can similarly be achieved by detecting if any signal reaches that point, if it does then one of the processors must have sent a signal and the value TRUE can be broadcast, if not the value FALSE is broadcast.

1.2.2 Traditional representation parallel arithmetic

Parallel Karatsuba

In [7], some experiments were performed using SugarBush. SugarBush is a parallel version of Maple, which uses C/Linda for communication. Using the Karatsuba algorithm the calculation of the first million digits of $\sqrt{2}$ was performed. Using 27 distributed workstations a speedup of 15 times was achieved, Although it was commented, that this was the best speedup achieved, with most runs taking significantly longer. It was proposed that this was due to network load.

While it is relatively easy to achieve good parallel speedup using numbers containing in excess of 100,000 bytes, it is much more interesting to look at what size of number is needed for parallel speedup. This will depend on the number of processors but for 3, 9 and 27 processors, speedup was achieved at 512, 1024 and 2048 decimal digits. (Note that Maple uses a number base of 10^n and that the Karatsuba algorithm creates 3 sub-multiplications at each step). To reach 50 percent parallel efficiency (e.g 1.5 times faster on 3 processors) numbers of size 1024, 8000 and 64000 decimal digits were required.

CALYPSO

Another implementation of a parallel Karatsuba algorithm was presented in [6], this was part of the PhD thesis of Giovanni Cesari [5]. In this thesis is a description of CALYPSO a parallel library for bignum arithmetic.

The CALYPSO library uses three different multiplication algorithms Classical, Karatsuba and FFT. It also uses both the Classical division, and Newtonian Inverse based division, which uses Karatsuba or FFT multiplications. Several different parallel platforms are tested but the majority of results come from a Cray T3D, and an Intel Paragon.

The parallel Karatsuba algorithm was tested using 3, 9, 27, and 81 processors on a Cray T3D. Table 1.1 shows the size of arguments in bytes which are required to achieve certain parallel efficiencies.

For numbers of over 2048 bytes, sequential tests showed that an FFT multiplication was faster than a Karatsuba multiplication. A parallel implementation of this algorithm was tested using numbers of processors equal to powers of 2. Speedup was achieved using up to 128 processors with inputs of size 1024 bytes.

	Number of Processors			
Efficiency	3	9	27	81
Break Even	512	512	512	1,024
50%	512	2,048	8,192	32,768
90%	4,096	32,768	131,072	524,288

Table 1.1: Size in bytes for stated efficiency of parallel Karatsuba algorithm

Using 128 processors fifty percent parallel efficiency was achieved with inputs of size 1 Megabyte, though with 4 processors a similar efficiency was achieved using the smallest numbers tested which were of length 1024 bytes.

In sequential tests, the Classical division algorithm was shown to be faster than a FFT based Newtonian Inverse for a dividend of 4096 bytes and a divisor of 2048 bytes. In parallel the same was true. Past this point the Newtonian division was faster in parallel using 16 processors.

The efficiencies of the parallel Newtonian division compared with a sequential Newtonian division was similar to those for the underlying FFT multiplications. With a 1 Megabyte dividend being divided by a 0.5 Megabyte divisor being 41 percent efficient on 128 processors.

Parallel Jebelean Exact Division

The ideas of Jebelean in [24] are taken one step further in [27] when the idea of calculating the least and most significant bits independently is explored. This effectively halves the sizes of the numbers involved giving a 2 times speedup.

As a single exact division becomes two independent calculations these could be performed on different processors potentially halving the execution time.

1.2.3 Arithmetic Hardware

Much of the literature on parallel arithmetic comes from circuit design. In a circuit it is possible to perform any number of logical operations at the same time. The time taken to perform an operation will depend on the number of gates a signal will need to pass through, this is referred to as the circuit depth. In this subsection the problem of addition and multiplication will be discussed, both of these can be realised with circuits of depth $O(\log n)$ where n is the size

of the inputs in bits.

While most division circuits use an efficient multiplication circuit to calculate a Newtonian inverse this is not optimal in terms of circuit depth. A depth $O(\log n)$ multiplication circuit would need to be used $O(\log n)$ times to calculate a Newtonian Inverse. If the circuit is being used to perform RSA style calculations a Montgomery reduction could be used. However it is possible to perform a general division with a circuit of depth $O(\log n)$ by using a modular representation.

Addition

If the Classical addition algorithm, Algorithm 1, were used in an addition circuit it would have a depth of $O(n)$. To be able to compute the i^{th} bit the carry from the $i-1^{th}$ bit is required.

To reduce the depth of the circuit it is necessary to calculate the carry in advance. This can be done by means of a Carry Lookahead Circuit [23]. It is assumed that the two numbers to be added are represented by the sequences of bits $\langle x_{n-1}, x_{n-2}, \ldots x_0 \rangle$, and $\langle y_{n-1}, y_{n-2}, \ldots y_0 \rangle$ and that the final answer is stored in $\langle z_n, z_{n-1}, \ldots z_0 \rangle$. The addition can be broken into three steps. Note $\oplus$ represents exclusive or.

- $u_i = x_i \wedge y_i, v_i = x_i \oplus y_i$ for $0 \leq i < n$
- Calculate c_i the carry bit for $0 \leq i < n$
- $z_0 = v_0, z_i = v_i \oplus c_{i-1}$ for $1 \leq i < n, z_n = c_{n-1}$

It is clear that the first and last steps can be carried out using circuits of constant depth, but the second step needs more explanation. Each carry bit relies on the previous one according to the following relation.

$$c_i = A(u_i, v_i)(c_{i-1}) = u_i \vee (v_i \wedge c_{i-1}) \tag{1.13}$$

By substitution, this can be turned into a relationship to the $i-2^{th}$ carry bit.

$$c_i = u_i \vee (v_i \wedge (u_{i-1} \vee (v_{i-1} \wedge c_{i-2}))) \tag{1.14}$$

$$c_i = (u_i \vee (v_i \wedge u_{i-1})) \vee ((v_i \wedge v_{i-1}) \wedge c_{i-2}) \tag{1.15}$$

$$c_i = A(u_i \vee (v_i \wedge u_{i-1}), v_i \wedge v_{i-1})(c_{i-2}) \tag{1.16}$$

In general the equation for c_i can be thought of as:

$$c_i = (A(u_i, v_i) \circ A(u_{i-1}, v_{i-1}) \circ \ldots \circ A(u_0, v_0))(0) \tag{1.17}$$

$$A(u_a, v_a) \circ A(u_b, v_b) = A(u_a \vee (v_a \wedge u_b), v_a \wedge v_b). \tag{1.18}$$

A binary tree can be constructed to calculate each carry bit in $\log_2 n$ depth where at each node the function composition $\circ$ is performed on a pair of functions. As the composition can be performed in constant depth this will lead to an overall depth for addition of $\log n$.

As an n input binary tree has size $O(n)$, and there are n carry bits to calculate this could lead to a size of $O(n^2)$. However, by combining parts of each tree this can be reduced to $O(n)$, leading to an addition circuit of optimum size $O(n)$ and optimum depth $O(\log n)$.

Multiplication

Using the Classical multiplication algorithm, Algorithm 4, the multiplication of two binary numbers would become the addition of n, $2n$ bit numbers. For example $\langle 1,0,0,1\rangle * \langle 1,0,1,1\rangle$ becomes:

$$\begin{array}{ll} 01001000 & + \\ 00000000 & + \\ 00010010 & + \\ 00001001 & . \end{array} \tag{1.19}$$

If a binary tree was used to add these numbers using the addition circuit described in the previous section the circuit would have size $O(n^2)$ and depth $O(\log^2 n)$. As stated before it is possible to reduce the depth of the circuit to $O(\log n)$. One method of doing this is to use a tree of Carry Save Adders [23].

A Carry Save Adder takes three numbers as input and gives two numbers as output. The sum of the three inputted numbers is equal to the sum of the two outputted numbers.

A Carry Save Adder is a simplified addition circuit. Instead of propagating the carry bits in the addition, they are saved in a separate number. If the inputs are X, Y and Z, and the outputs are U and V, then U will be the result of the

addition with the carries ignored, and V will be made up of all the carries. This is put more formally below.

- $u_i = x_i \oplus y_i \oplus z_i$, for $0 \leq i < n$, $u_n = 0$
- $v_0 = 0, v_i = (x_{i-1} \wedge (y_{i-1} \vee z_{i-1})) \vee (y_{i-1} \wedge z_{i-1})$, for $1 \leq i \leq n$

Each Carry Save Adder has size $O(n)$ and depth $O(1)$. A tree of such adders would have a depth $\log_{3/2} n = O(\log n)$, and size of $O(n^2)$. The output of the tree will be two numbers which can then be added together using a normal addition circuit which has size $O(n)$ and depth $O(\log n)$.

Montgomery Reduction

If you have a large number of operations to perform, or your input data is available one bit at a time, a systolic array can be used. A systolic array is an array of inter-connected cells. Each cell can communicate only with its neighbours. While they can be of any dimension, it is common to use either a one dimensional line of cells, or a two dimensional square of cells.

The systolic arrays get their name from the beating of the heart. This is because data flows through the array. During each step each cell gives an output, which depends on its current state, and the inputs from other cells.

While systolic arrays for multiplication are well known [26], with Atrubin proposing a linear array in 1965, more recently systolic arrays for Montgomery Reductions have been proposed. In [14], a one dimensional array is proposed which can perform an n bit reduction in $2n + 4$ cycles, using n cells.

In [40], a two dimensional array is proposed. This takes $2n + 2$ cycles to perform the first reduction, but thereafter can produce one answer per cycle. For a large number of inputs this reduces the time to $O(1)$ using n^2 cells.

Division

The Classical division algorithm, Algorithm 5, is not a good candidate for a $O(\log n)$ depth circuit as it consists of $O(n)$ steps each dependent on the last. A Newtonian inverse based division is also not suitable as calculating the inverse will take $O(\log^2 n)$ time. It is, however, possible to produce a $O(\log n)$ depth circuit but a completely different approach must be taken.

The first published $O(\log n)$ depth division circuit was [2]. The circuit had size $O(n^4 \log^3(n))$. Another depth $O(\log(n))$ circuit is described in [8]. This has size $O(n^6/\log(n))$ but was designed in a way which would make it easier to manufacture.

In both of these papers, the traditional representation of numbers has been abandoned. Instead each number is stored as a set of residues to relatively prime moduli. While neither of these circuits would be practical for implementing on a parallel computer, due to the circuit size, storing numbers in this way has many advantages for parallel computing and is the topic of Section 1.3.

1.3 Modular representation

1.3.1 Basic Modular arithmetic

First we must define what it means for two numbers to be congruent ($\equiv$) to a positive modulus M.

$$A \equiv B \bmod M \Rightarrow M|A - B \tag{1.20}$$

If $A \equiv A' \bmod M$ and $B \equiv B' \bmod M$, then the following equivalences hold.

$$A + B \equiv A' + B' \bmod M \tag{1.21}$$

$$A - B \equiv A' - B' \bmod M \tag{1.22}$$

$$A * B \equiv A' * B' \bmod M \tag{1.23}$$

Because the above equivalences hold it is possible to perform modular arithmetic using *least residues*, which we will represent as $|A|_M$. These are defined such that $|A|_M \equiv A, 0 \leq |A|_M < M$. (As noted in [17] it would be more precise to call these least positive residues).

Another important concept in modular arithmetic is the inverse. If A and M are relatively prime then there exists B such that $A.B \equiv 1 \bmod M$. B is called the inverse of A and is written as $A^{-1} \bmod M$, or $|A^{-1}|_M$ as a least residue. This can be used to perform exact division as is shown in the following equivalence.

$$B|A \Rightarrow A/B \equiv A * B^{-1} \bmod M \tag{1.24}$$

1.3.2 Common uses of modular arithmetic

Modular arithmetic is widely used in computer algebra. In [38], modular algorithms are split into three variants. These are *big prime*, *small primes* and *prime power*.

In the big prime variant, all calculations are carried out modulo a single prime number. If a bound on the size of the answer is known, then by choosing a prime which is at least as big as this bound, the answer modulo the large prime will equal the true answer.

In the small primes variant, all calculations are carried out modulo several different small primes. Again, it is assumed that a bound on the answer is known. This time, it is the product of the small primes that must be larger than the bound. Using several small primes is usually faster than using one large prime, as modular calculations usually have time complexities which are greater than linear in the number length. At the end of the calculation a Chinese Remainder Theorem reconstruction is used to form the final answer.

In the prime power variant, a single small prime modulus is used. However, this time, the answer modulo p^n is used to calculate the answer modulo p^{n+1}. This is known as lifting. As with the other two variants it is assumed that a bound on the answer is known, the modulus being increased until it is greater than this bound.

Not all calculations can be carried out using modular algorithms, as there are no modular equivalents of comparison or general division. To be able to perform either of these operations, the whole number needs to be considered not just part of it. As only the answer is bounded in the above algorithms intermediate expressions could be much larger. If this was not the case the large prime variants would be of limited use.

In subsection 1.3.3, a modular representation is described which considers a number as a set of residues much as in the small prime methods described above. However, it is now assumed that the number represented is less than the product of the moduli. With this restriction, it becomes possible to perform comparison

and general division.

1.3.3 Definition of a modular representation

In a modular representation a number is stored as a sequence of residues. Each residue corresponds to a different modulus, each of which is relatively prime to all of the others. If the moduli are $p_0, p_1, \ldots, p_{n-1}$, then an integer X would be represented as $\langle x_0, x_1, \ldots, x_{n-1} \rangle$ where $x_i \equiv X(\mathrm{mod} p_i)$, $0 \leq x_i < p_i$. The product of all the moduli is $M = p_0 p_1 \ldots p_{n-1}$.

The Chinese Remainder Theorem (CRT) states that the system of simultaneous congruences,

$$X \equiv x_i \bmod p_i, 0 \leq i < n-1 \tag{1.25}$$

has a unique solution $\bmod p_0 p_1 \ldots p_{n-1}$, assuming $\gcd(p_i, p_j) = 1$, for $i \neq j$. Hence, each of the numbers between 0 and $M-1$ can be uniquely represented.

1.3.4 Previous Work

Introduction

The use of a modular representation dates back many years. In 1967, Szabo and Tanaka published the book *Residue arithmetic and its applications to computer technology* [37]. This covered much of the literature on the subject, and details solutions to the problems of comparison, sign detection, base extension and division.

Later works on the subject often refer to a modular representation as a Residue Number System (RNS). In contrast, Knuth in [26] simply refers to Modular Arithmetic. It is these more recent publications which will be described now.

Approximate Reconstructions

It is not possible to determine the magnitude of a number, stored as a set of residues, without considering all the residues. Indeed, without considering all the residues, it is not even possible to approximate the magnitude of a number. For example, working with the list of moduli $\langle 2, 3, 5 \rangle$, the partial list of residues $\langle 1, 1, ? \rangle$, could represent 1, 7, 13, 19, or 25. To reconstruct the number exactly a CRT(Chinese Remainder Theorem) reconstruction is used, this is described in

subsection 2.2.1. However, this involves a great deal of computation and the ability to handle larger integers.

It is, however, possible to form an approximation of the CRT reconstruction using numbers of bounded precision. This is the topic of Section 2.2. It is possible to form this approximation with a circuit consisting of n adders each handling $O(\log(n))$ bits and a table of size $O(n2^b \log(n))$, where n is the number of moduli and 2^b is greater than any of the moduli [20]. This circuit has a depth of $O(\log(n))$ and as such is optimal. A similar circuit is described in [39] where a 'fractional representation' is referred to, and also in [18].

As the approximate reconstruction is an estimate of the true value, errors are inevitably involved. In [21] it is shown that the worst case error for an approximate reconstruction is the value shown in Equation 1.26, where d is the number of bits of precision.

$$-2^{-d} \sum_{0 \leq i < n} \frac{m_i - 1}{m_i}. \tag{1.26}$$

A more detailed discussion of the errors involved in an approximate reconstruction can be found in subsection 2.2.3.

An alternative approach to the approximate reconstruction is proposed in [12]. The diagonal function of a number X, $D(X)$ is defined as:

$$D(X) = \sum_{i=0}^{n-1} \lfloor X/p_i \rfloor. \tag{1.27}$$

Due to the precise nature of its definition $D(X) < D(Y) \Rightarrow X < Y$ and $D(X) > D(Y) \Rightarrow X > Y$. However $D(X) = D(Y)$ only tells us that X and Y are of similar size. While quick and accurate when $n = 2$, as the number of moduli increase it becomes less effective.

Base Extension

Another method, which can be used to gain information about the whole list of residues, is to increase the length of the list. This is known as base extension. There are two main methods used to achieve this both of which are detailed in Section 2.4.

The first method, Mixed Radix Conversion (MRC), is detailed in [37] and

described in subsection 2.3.2. In [19], a circuit is described which can perform an MRC with a depth of $O(\log n)$, similar circuits are used in [1] and [13].

The second method is to reconstruct the new moduli using the constants formed in an approximate reconstruction. Instead of performing an approximate reconstruction using limited precision, it is possible to perform a reconstruction to a new modulus. As each of the reconstruction can be done in parallel, this again leads to a circuit depth of $O(\log n)$ as is described in [31]. Details of this reconstruction can be found in subsection 2.2.6.

Sign Detection

If the product of all the moduli is M then it is possible to represent all values of X, such that $0 \leq X < M$. This is useful if X is known to be positive, but for more flexibility restricting X to the range $-\lceil M/2 \rceil \leq X \leq \lfloor M/2 \rfloor$ is common. Once this has been done the problem of comparison can be reduced to a sign detection by subtracting one number from the other.

Two approaches can be taken to sign detection: either, perform a base extension/MRC to gain enough information to determine the sign in a single operation, or use a sequence of approximations until the sign can be determined with certainty.

To determine sign in [1] the following calculation is performed.

$$sign(X) = \frac{2X - |2X|_M}{M} \tag{1.28}$$

This will lead to a figure of either 0 or 1. If it is 0 then the number is positive, if it is 1 it is negative. To read this value a single extra modulus is employed. To be able to calculate this modulus is non-trivial however, as a mixed radix conversion is employed.

In [13], the sign of a determinant is calculated using the base extension method described in [19]. By calculating the answer using a set of moduli twice as large as is necessary, it is then possible to check the sign by base extending half of the residues assuming the answer is positive. If the extra residues match in both cases then the number is positive, otherwise it is negative.

While most of the recent research has been in the field of circuit design, one paper [3] looked at the problem of sign detection using a single processor

machine. On a single processor, base extension becomes an $O(n^2)$ operation, whereas an approximate reconstruction is an $O(n)$ operation. In most cases the sign of a number can be determined using a single approximate reconstruction saving a significant amount of time. This paper details several different methods for sign detection all of which both appear both in Section 2.3 and in the author's own paper [33], which was submitted before the publication of [3]. The paper mentions several applications of sign detection including the calculation of Sturm sequences for isolating algebraic roots of polynomials, and calculating the sign of matrix determinants as in [13].

Division by known divisor

With the rise in popularity of the RSA cryptography, many authors have looked to performing modular reductions using large moduli. One popular approach is to use the Montgomery reduction described in subsection 1.1.4. Circuits to perform this reduction were proposed in [32] and [35].

Both papers follow the same method, in order to perform the Montgomery reduction a modulus is chosen in which it is easy to perform calculations. In our case the natural choice is M, but this leads to two problems, the sub-result is larger than M and an exact division by M is required. As mentioned above exact division can be performed by calculating an inverse and multiplying. However, to calculate an inverse the number and the modulus must be relatively prime.

To get around both of these problems an extended set of moduli are used called $\overline{M}$. Using a normal base extension the effective modulus becomes $M\overline{M}$. It is then possible to perform the exact division in $\overline{M}$, as it has to be relatively prime to M to be a valid extension. To get the final result a reverse base extension is used to recreate the answer modulo M. The process is then repeated using the normal method for performing RSA calculations.

In [22] two alternative methods are considered to perform division by a known divisor which can then be used to perform RSA by subtracting the quotient multiplied by the divisor.

In the first method, the value $\lfloor M/N \rfloor$ is calculated using an already existing division algorithm such as in [20]. By multiplying the number to be divided by $\lfloor M/N \rfloor$, a number which is approximately M times too much is formed. If M is large enough, then the error after division by M will be at most one. The difficult

part of this method is the now inexact division by M.

To perform an inexact division by M again a second set of moduli, the product of which is $\overline{M}$, is introduced. By performing a base extension on the number modulo M, the remainder part can be calculated. Once this is known it can be subtracted off and an exact division by M can carry on as above.

The second method proposed in [22] uses the approximate reconstruction constants to reconstruct to a large modulus, much as in the second base extension method described above. To save time many of the sub-products can be pre-calculated, to reduce the problem to one of scaling (multiplying by a single digit constant), addition and then a small general division using the method in [20].

General Division

As mentioned in subsection 1.2.3 it is possible to build a division circuit which uses a modular representation and has a depth of $O(\log n)$, however the size of such circuits is too large to be of interest in a practical parallel computing context. Two papers have described circuits which can perform general division using more modestly sized circuits.

In [20] an approximate reconstruction is used to estimate the sign of a number. The estimate of the sign was either definitely positive, definitely negative or unknown. However, if the sign was unknown it was certain to be small compared to the modulus M. Using this sign estimate the quotient can be built up bit by bit. Using n adders of width $O(\log n)$ and tables of size $O(n2^b \log(n))$, where 2^b is greater than any one modulus, the circuit performs division in time $O(nb \log(n))$ in un-optimised form or $O(nb)$ when optimised.

In [19], a general division method base on a Newtonian inverse is described. Instead of using an existing general division method to calculate the inverse an iteration is used as shown in Equation 1.29.

$$Z_{i+1} = \left\lfloor \frac{Z_i(2M - NZ_i)}{M} \right\rfloor \tag{1.29}$$

A division by M is required in every step of this iteration, which given a reasonable first estimate will take $O(\log n)$ steps. Assuming a size $O(n^2)$ base extension circuit of depth $O(\log n)$. The general division will take time $O(\log^2 n)$ to perform a general division using a circuit of size $O(n^2)$.

1.4 Thesis Structure

The ultimate aim of the research described in this thesis, is to develop a library of functions to allow bignum calculations to be performed on a parallel computer.

The library uses a modular representation, as it is well suited to a parallel computer. Previous work on using a modular representation is described in this Chapter.

The library is based around a few fundamental algorithms, which are described in Chapter 2. The two most fundamental algorithms, are an approximate CRT reconstruction, and a reconstruction to a small modulus. The algorithm for reconstructing to a small modulus being a new result.

A new definition of number length is also described in Chapter 2 and three algorithms are proposed to calculate it. Two of these, repeated approximations and a probabilistic binary search, are also new results.

The last section of Chapter 2 deals with increasing the number of residues used to store a number, three algorithms are described to perform this task, two of which use the reconstruction to a small modulus.

In Chapter 3, a library of functions is described in which the modular representation is paired with an approximation of the size of the number. Using this representation all the algorithms needed to perform general arithmetic are described. Because we have an approximation of the size of the number and can extend the set of moduli dynamically, we can, for the first time in multi-modulus methods, provide comparison and general (quotient & remainder) division.

In Chapter 4, a different datatype is described, in which the number is stored using a fixed number of residues without any approximation of the size of the number. Using this datatype, a more traditional style of modular calculation can be performed. An implementation of the probabilistic length algorithm from Chapter 2 is also described in this chapter.

The methods used to test the library are presented in Chapter 5. A wide range of hardware is used to perform these tests and this is first described. One set of tests, are used to benchmark the three core functions of the library. A second set of tests, are performed to test the speed and correctness of the arithmetic operations of multiplication, division and addition.

Of all the arithmetic operations, general division is by far the most costly.

Chapter 6 describes two methods of avoiding this operation. One method is to first check to see if the division is exact using a divisibility test. A new probabilistic divisibility test is presented which is significantly faster than deterministic tests. The other method uses a specialist base extension, to perform division by a known divisor. This is the first implementation of this method which was originally intended as a circuit design.

Two examples of the library in action are given in Chapter 7. The first example is calculating GCDs, a common operation in bignum calculations. The second example is calculating the determinant of an integer matrix. In this second example calculations are performed using both the standard datatype and the datatype described in Chapter 4.

Finally, in Chapter 8, conclusions are drawn, and further work is discussed.

Chapter 2

Fundamental algorithms using a modular representation

2.1 Introduction

In this chapter, the most fundamental algorithms will be presented. These algorithms deal with a general number X which is stored as a list of residues $\langle x_0, x_1, \ldots, x_{n-1} \rangle$ to a known list of moduli $\langle p_0, p_1, \ldots, p_{n-1} \rangle$.

None of these algorithms assume any additional information about X is known. This is in contrast with Chapter 3 in which the algorithms assume that values such as $len(X)$ and k have already been calculated. The meanings of these terms will be discussed in this chapter.

In Section 2.2, an *Approximate CRT Reconstruction* is described. This reconstruction gives an approximation of the fraction X/M, where M is the product of all moduli $p_0 p_1 \ldots p_{n-1}$. Both the sequential and parallel implementation of this reconstruction is discussed, along with the error inherent in the calculation. Also discussed are the ambiguities which can be caused by these errors, and how floating point arithmetic can be avoided.

A *Reconstruction to a Small Modulus* is also described in Section 2.2. This allows the calculation of X mod m, for a small modulus m.

In Section 2.3, the concept of length in a modular representation is introduced. Three methods are described to calculate this length. The first uses a *Mixed Radix Conversion* to speed up the reconstruction of the modular num-

ber, this has certain advantages over a full *CRT Reconstruction* including ease of parallelism. The second, *Repeated Approximations*, uses a number of *Approximate CRT Reconstructions* to find the length of the number. The final method, *Probabilistic Binary Search*, involves a binary search coupled with a probabilistic confirmation function. In different situations each of these could be the most appropriate, so these situations are discussed.

Section 2.4 concerns increasing the number of residues used to represent X. Three methods are discussed. The first *Individual Reconstructions*, is suitable for a small increase in the number of moduli. The second, *Mixed Radix Extension* uses the *Mixed Radix Conversion* discussed in subsection 2.3.2 for larger increases in the number of residues. The third, *Combined Individual Reconstructions* combines several *Individual Reconstructions* in a way that decreases both the amount of calculation and communication.

Throughout this chapter comment will be made on the parallel aspects of these algorithms. In particular, the role that broadcasts, reductions and globalor operations have is discussed where it is appropriate.

2.2 Approximate CRT Reconstruction

2.2.1 The Chinese Remainder Theorem

The Chinese Remainder Theorem states that the system of simultaneous congruence,

$$\begin{aligned} X &\equiv x_0 \pmod{p_0} \\ X &\equiv x_1 \pmod{p_1} \\ &\vdots \quad \vdots \quad \vdots \\ X &\equiv x_{n-1} \pmod{p_{n-1}}, \end{aligned} \tag{2.1}$$

has a unique solution $\bmod p_0 p_1 \ldots p_{n-1}$, assuming $\gcd(p_i, p_j) = 1$, for $0 \leq i, j < n$, $i \neq j$.

To show that any solution would be unique, we can consider two numbers A and B that obey the congruences of Equation 2.1. The difference $A - B$ must be divisible by all p_i, and as these are relatively prime, the difference must be a

multiple of their product and hence $A \equiv B \bmod p_0 p_1 \ldots p_{n-1}$.

That a solution exists can be shown by considering the following equation,

$$Crt(X) = \sum_{i=0}^{n-1} M_i y_i \tag{2.2}$$

$$\text{where } M = \prod_{i=0}^{n-1} p_i, M_i = \frac{M}{p_i} \text{ and } y_i \equiv M_i^{-1} x_i \pmod{p_i}.$$

To show that Crt(X) obeys each of the congruences of Equation 2.1 we can consider the case of an arbitrary p_i. As $M_j \equiv 0 \bmod p_i$ whenever $i \neq j$, then Equation 2.2 can be reduced to the single term,

$$Crt(X) \equiv M_i M_i^{-1} x_i \equiv x_i \pmod{p_i}$$

therefore $Crt(X) \equiv X \bmod p_i$, for all p_i and hence $Crt(X) \equiv X \bmod M$.

Computational cost of reconstruction

The costs of the reconstruction can be split into two parts: those which depend on the set of moduli being used and those which also depend on the value of X. If several reconstructions are being made using the same set of moduli, it will save time to store the first set of values.

If it is assumed that each modulus is slightly smaller than the size of the maximum allowed using the machine integer type, then the number of machine integers needed to store M will be n, for each M_i it will be $n-1$ and for each $Inv_i = M_i^{-1} \bmod p_i$ it will be 1. The space needed to store these values will be $O(n^2)$.

Assuming these values have been pre-calculated the cost of calculating each y_i will be independent of the number of moduli, as x_i and Inv_i are of fixed size. To form each $M_i y_i$ will involve a scaling operation which will take a time proportional to the length of M_i. The summation will involve n additions of n digit numbers, with a comparison with M and possible subtraction at each step, or alternatively a reduction mod$M$. This gives a sequential time complexity of $O(n^2)$.

2.2.2 Approximate CRT Reconstruction

The approximate reconstruction presented here has appeared in several papers on circuit design including [20],[39] and [18]. For more details on the contents of these papers see subsection 1.3.4.

In an approximate reconstruction, instead of calculating the exact value of X, an approximation is made of the fraction X/M. Taking Equation 2.2 as a start point an exact value of X/M can be calculated as follows.

$$\sum_{i=0}^{n-1} M_i y_i \equiv X \pmod{M}$$

$$\sum_{i=0}^{n-1} M_i y_i = kM + X, k \in \mathcal{Z} \tag{2.3}$$

$$\sum_{i=0}^{n-1} \frac{y_i}{p_i} = k + \frac{X}{M} \tag{2.4}$$

To obtain an approximation of this summation, each of the fractions y_i/p_i are first calculated to a fixed precision and then they are summed. The integer part of this summation will be k, and the fractional part will be X/M.

Computational cost of reconstruction

To perform the approximate reconstruction the values of M and M_i are not needed, instead only the values Inv_i are required. This reduces the space complexity of the stored tables to $O(n)$.

The time needed to calculate y_i/p_i will depend on the precision required but will always be independent of n. As the size of these approximations is fixed, the time complexity of summation will be $O(n)$.

When performing this reconstruction in parallel, each of the approximations can be calculated independently as the value of y_i/p_i is dependent only on x_i, Inv_i and p_i. Each processor can perform a local summation followed by a parallel summation.

Each processor only requires the values of Inv_i and p_i which are associated with the y_i/p_i being calculated. This allows the tables to be divided up between the processors.

If there are *nproc* processors, the space required to store the values of Inv_i will be $O(n/nproc)$ per processor, and the time taken will be $O(n/nproc)$ plus the cost of the parallel summation. Which as was shown in subsection 1.2.1 is $O(\log(nproc))$.

2.2.3 Errors in an approximate CRT reconstruction

The errors inherent in an approximate CRT reconstruction were detailed by Hung and Parhami in [21]. Below an alternative derivation of the two main results is presented. This alternative derivation will be used in the next section on integer approximations.

Errors in the calculation of y_i/p_i

In calculating the total error in an approximate reconstruction, it is first necessary to calculate the error in each approximation of y_i/p_i. If d bits of precision are used to calculate each fraction, then the estimate will be $E_i/2^d$, where E_i is defined in Equation 2.5.

$$E_i = \left\lfloor \frac{2^d y_i}{p_i} \right\rfloor \tag{2.5}$$

The error, e_i, in this estimation must be less than $1/2^d$ by the definition of E_i.

$$e_i = \frac{y_i}{p_i} - \frac{E_i}{2^d} = \frac{y_i 2^d - E_i p_i}{p_i 2^d}, 0 \leq e_i < 1/2^d \tag{2.6}$$

If $m_i = p_i 2^d e_i$, then m_i will be an integer obeying the following.:

$$m_i = y_i 2^d - E_i p_i, 0 \leq m_i < p_i \tag{2.7}$$

$$m_i \equiv y_i 2^d \bmod p_i, 0 \leq m_i < p_i \tag{2.8}$$

As m_i can only take at most p_i values and is defined by a congruence modp_i, its value is unique, so the error in estimating y_i/p_i using d digits of base 2 is $m_i/p_i 2^d$, where m_i obeys Equation 2.8.

Worst case numbers

If summation errors are ignored, the total error in an approximate CRT reconstruction is defined by Equation 2.9.

$$\sum_{i=0}^{n-1} \frac{m_i}{p_i 2^d} \tag{2.9}$$

The largest error will occur when $m_i = p_i - 1$, for all i. In this case the error will be

$$\sum_{i=0}^{n-1} \frac{p_i - 1}{p_i 2^d}. \tag{2.10}$$

For this to occur the values of the x_i needed will be those which cause m_i to be equal to $p_i - 1$.

$$m_i \equiv p_i - 1 \bmod p_i \tag{2.11}$$

$$y_i 2^d \equiv -1 \bmod p_i \tag{2.12}$$

$$x_i Inv_i 2^d \equiv -1 \bmod p_i \tag{2.13}$$

$$x_i \equiv -1/(Inv_i 2^d) \bmod p_i \tag{2.14}$$

$$x_i \equiv -M/(p_i 2^d) \bmod p_i \tag{2.15}$$

Note that the largest error possible as defined in Equation 2.10 is bounded by $n/2^d$, which will be used for the sake of simplicity.

2.2.4 Integer approximations

The previous error analysis relied on the fact that no extra errors would be introduced by summing the approximated fractions. If floating point numbers are used for this summation this will not be the case. Furthermore, the use of floating point summation will produce different results if the summation is performed in a different order. Because of these disadvantages, it is desirable to perform the calculations using integers.

Below is presented a new method for calculating integer approximations. This method can be easily extended to calculate approximations to any desired accu-

racy using integers of bounded size.

Calculating E_i

Given m_i calculated using Equation 2.8 above, it is possible to calculate E_i without needing to use floating point calculations. Rearranging Equation 2.7, E_i can be calculated directly as,

$$E_i = \frac{y_i 2^d - m_i}{p_i}. \tag{2.16}$$

Assuming $0 \leq y_i < p_i$ then $0 \leq E_i < 2^d$, so E_i can be calculated by the congruence.

$$E_i \equiv \frac{y_i 2^d - m_i}{p_i} \equiv \frac{-m_i}{p_i} \bmod 2^d \tag{2.17}$$

Note that, as powers of 2 are being used, the evaluation of the congruence will require no explicit remainder operations.

Greater accuracy

Having calculated E_i and m_i it is possible to increase the accuracy of the calculation by repeating the process with m_i in the place of y_i.

$$\frac{y_i}{p_i} = \frac{E_i}{2^d} + \frac{m_i}{p_i 2^d}, 0 \leq m_i < p_i \tag{2.18}$$

$$\frac{m_i}{p_i} = \frac{E_i'}{2^d} + \frac{m_i'}{p_i 2^d}, 0 < m_i' < p_i \tag{2.19}$$

$$\frac{y_i}{p_i} = \frac{E_i}{2^d} + \frac{E_i'}{2^{2d}} + \frac{m_i'}{p_i 2^{2d}}, 0 < m_i' < p_i \tag{2.20}$$

This will double the accuracy of the estimation without needing to revert to larger number formats than we wish to use. For example, to get 64 bits of precision, 32 bit numbers can be used for the calculation.

2.2.5 Calculating the value k

The value of k is defined in Equation 2.3. While the value of k is simple to calculate in a full CRT reconstruction, in an approximate reconstruction some

ambiguity can be caused by errors in the approximations. Several different solutions have been proposed to avoid this ambiguity, the simplest of which is presented here.

To illustrate the problem, imagine the summation in Equation 2.4 came to 7.99 and the possible error was 0.02. In this case, either the value of X is slightly smaller than M and the value of k is 7, or the value of X is small compared to M and the value of k is 8. This is represented in Figure 2-1. Note that in the diagram the fractional part of the summation is represented by its position on the circle, the value of k is not represented.

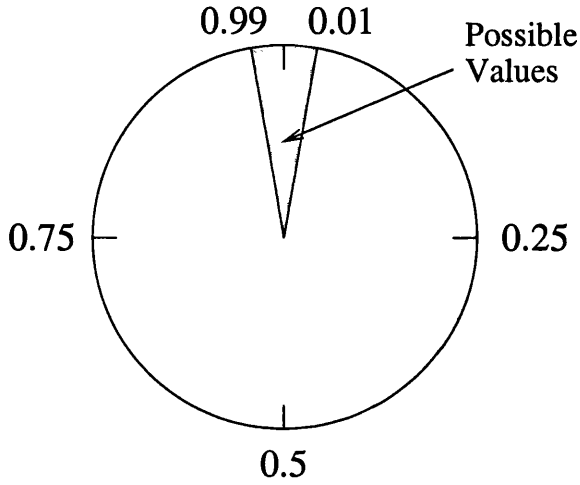

Figure 2-1: Unknown value of k

One simple method to remove this ambiguity is to restrict the value of X such that $0 \leq X < M(1 - max\_error)$. Once this is done it will be known that any estimate greater than $1 - max\_error$ will involve increasing the value of k by 1. This is demonstrated in Figure 2-2.

2.2.6 Reconstructing to a Small Modulus

Below a reconstruction to a small modulus is presented. In [31] a similar method was described for use in base extensions. The method below was however developed independently and was presented as such in [33].

While the approximate reconstruction gives information about the most significant digits of a number, the *Reconstruction to a Small Modulus* gives precise

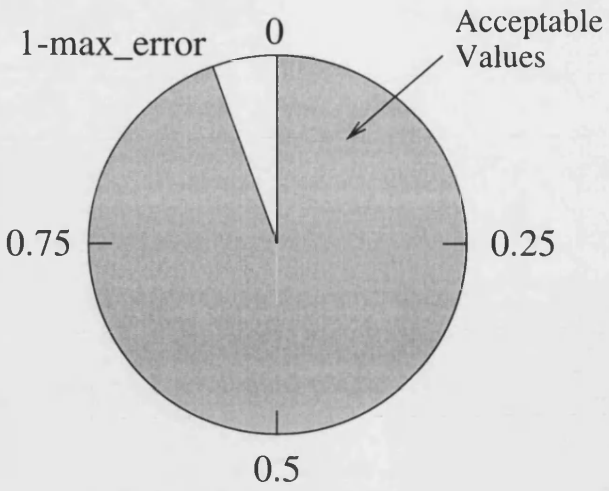

Figure 2-2: Acceptable values for X

detail about the number as a whole. It is essential that the value of k is known as this is used during the reconstruction.

The value of X is reconstructed to the modulus m. It is assumed that $\gcd(m, M) = 1$, and that m is sufficiently small to allow calculations $\bmod m$ to be performed in unit time. Considering again Equation 2.4 the summation can be rewritten modulo m.

$$\sum_{i=0}^{n-1} \frac{M}{p_i} y_i = kM + X$$

$$X = \sum_{i=0}^{n-1} \frac{M}{p_i} y_i - kM$$

$$X \equiv M \left(\left(\sum_{i=0}^{n-1} p_i^{-1} y_i \right) - k \right) \pmod{m} \tag{2.21}$$

Computational cost of reconstruction

If it is assumed that k is already known, then there are two main stages of the calculation: the calculation of $M \bmod m$ and the inner product. To calculate $M \bmod m$ is another example of a reduction operation, as all the p_i are known, in each step the two sub answers are multiplied together and reduced $\bmod m$. This will take time proportional to the number of moduli.

To calculate each inner product will involve first knowing the values of y_i and $p_i^{-1} \bmod m$, y_i can be calculated from Inv_i and x_i as is described in subsection 2.2.2, $p_i^{-1} \bmod m$ can be calculated using the *Extended Euclidean Algorithm.* The time taken to calculate each of these inner products will be dependent on the size of m and not the the number of moduli n.

The inner summation will again be a reduction operation this time using addition and reducing mod m, again the total time taken will be proportional to the number of moduli.

The time complexity for the whole calculation is thus $O(n)$ sequentially and $O(n/nproc) + O(\log(nproc))$ in parallel, both assuming the size of m is small. While in complexity terms this is the same as an *Approximate CRT Reconstruction* the calculation of the inverses is significant.

Storing inverses

One method of avoiding the calculation of inverses is to store the values in tables. As the value of m is unknown it would be inconvenient to store all the inverses of all numbers less than a certain bound. Instead, it is simpler to store the inverses of the individual moduli. This will take a number of integers equal to the sum of all the moduli.

The value of $p_i^{-1} \bmod m$ can be calculated from $m^{-1} \bmod p_i$, using Equation 2.22.

$$p_i^{-1} \bmod m = m - \frac{(m^{-1} \bmod p_i)m - 1}{p_i} \tag{2.22}$$

2.3 Calculating the length of a number

2.3.1 Length in a modular representation

In this section a new definition of the length of a number stored in a modular representation is presented. As was commented in subsection 1.3.4, the calculation of length is closely related to the problem of sign detection.

In a traditional representation the length of a number is easily defined as the number of digits needed to represent the number. This is clear as any other digits

used beyond this length will be equal to zero. For example, when using 5 decimal digits 123 becomes 00123, the two leading zeros are clearly not needed.

In a modular representation, each of the residues is likely to be non-zero, irrespective of the number being represented (unless the number is zero when all the residues will be likewise). It is not clear if all the residues are needed to successfully reconstruct the number. Without this knowledge, it will be impossible to tell if the sum or product of two numbers will be larger than the modulus being used.

The simplest definition of length is the smallest number of residues, such that their product is greater than the number being represented. That is the length of X will be $len(X)$ as defined in Equation 2.23.

$$\prod_{i=0}^{len(X)-2} p_i \leq X < \prod_{i=0}^{len(X)-1} p_i \tag{2.23}$$

Note for this definition to be consistent with the previous statement it is important that the moduli are ordered such that $p_0 > p_1 > \ldots > p_{n-1}$.

2.3.2 Mixed Radix Conversion

Below is a method of calculating length based on a Mixed Radix Conversion. This is a new application of a method which is normally used for base extension.

The simplest method of determining the length of a number stored in a modular representation would be to perform a full CRT reconstruction. The number obtained could then be compared with the pre-calculated products of moduli, or estimated from the length of the number in a traditional representation. This is simple, but it is time consuming.

Another style of full reconstruction is a Mixed Radix Conversion or MRC. After performing an MRC, the output is a sequence of digits. The positions of these digits do not represent powers of a number base, but instead products of the moduli. At the end of the reconstruction, the number X will be represented by the sequence $\langle d_0, d_1, \ldots, d_{n-1} \rangle$, where $d_i < p_i$. Given this sequence of digits the value of X could be calculated according to the Equation 2.24.

$$\langle d_0, d_1, \ldots, d_{n-1} \rangle \rightarrow d_0 + \sum_{i=1}^{n-1} \left(d_i \prod_{j=0}^{i-1} p_j \right), 0 \leq d_i < p_i \tag{2.24}$$

Once X has been converted to this representation, then $len(X)$ will follow directly from the definition. It will equal the index of the greatest non-zero digit plus one, which is the same definition of length used for single radix numbers.

Calculating d_i

If Equation 2.24 is reduced mod p_0, only d_0 will remain, this is known to be less than p_0 and hence it must be equal to x_0 which is X mod p_0.

$$x_0 \equiv X = d_0 + \sum_{i=1}^{n-1} \left(d_i \prod_{j=0}^{i-1} p_j \right) \equiv d_0 \bmod p_0 \tag{2.25}$$

The value of d_1 can be obtained in a similar manner if $X - d_0$ is divided through by p_0.

$$\frac{x_1 - d_0}{p_0} \equiv \frac{X - d_0}{p_0} = d_1 + \sum_{i=2}^{n-1} \left(d_i \prod_{j=1}^{i-1} p_j \right) \equiv d_1 \bmod p_1 \tag{2.26}$$

In general the value of d_i can be defined as the following.

$$d_i = x_{i,i} \tag{2.27}$$

$$x_{i,m} = \begin{cases} x_i & \text{if } m = 0 \\ \frac{x_{i,m-1} - x_{m-1,m-1}}{p_{m-1}} \bmod p_i & \text{if } 1 \leq m \leq i \end{cases} \tag{2.28}$$

Note that the values of $x_{i,m}$ with $m > i$ are not needed as they will not affect the rest of the calculation.

Computational costs of MRC

If all the values of $x_{i,m}$ are calculated in step m, then the m^{th} step will involve $n - m - 1$ subtractions and modular divisions. To perform the modular division an inverse will need to be calculated as was the case for *Reconstruction to a Small Modulus* described in Section 2.2.6. Note that this time all of the inverses are of other moduli, thus they could be stored using n^2 integers as opposed to the

$\sum_{i=0}^{n-1} p_i$ integers needed to store a full set of inverses.

The calculation will end when all the remaining $x_{i,m}$ are zero. This will happen when $m = len(X)$. The total computational cost will thus be $O(n\, len(X))$.

In parallel two things will need to be communicated, the values of d_i and the fact that the calculation has finished. The obvious way to do this is to broadcast the value of d_m at the end of each step, and to check if the calculation has finished at the beginning of each step. The calculation is finished when $\bigwedge_{i=m}^{n}(x_{i,m} = 0)$. This is a reduction operation using AND. In the correct circumstances these operations can be performed in constant time and thus will not add to the complexity of the algorithm. If special hardware does not exist then the time taken will be the same as a reduction operation $O(\log(nproc))$.

Whatever the costs of communication, all the calculations involving a single modulus can be performed in isolation, so the parallel complexity becomes $O((n\, len(X))/nproc)$ plus a possible $O(len(X)\log(nproc))$ for communication.

2.3.3 Repeated approximation

Calculating the length of a modular number using repeated approximations is not based on any previous work. A similar method has since been proposed in [3] for use in sign detection on a sequential computer.

By using the *Approximate CRT Reconstruction* of Section 2.2 it is possible to obtain an estimate of X/M. This can then be used to investigate the length of a number. However, if the number being investigated is significantly smaller than M then it will have an approximation which is too small to be distinguished from the error in the approximation.

To obtain a more accurate estimate, the method described in subsection 2.2.4 could be used. This would be similar to using a full reconstruction instead of an MRC. The estimate would need to be compared to products of moduli which would involve calculations with numbers longer than a single integer in length.

Instead of using a repeatedly more accurate estimate using the original set of moduli, it is simpler to reduce the number of moduli used in the reconstruction. This will allow the same sized estimate to be used and also will effectively multiply the original estimate by the product of the moduli removed.

This method involves removing moduli from the reconstruction until the esti-

mate is so large, that to remove any more moduli would result in reconstructing to a modulus that is smaller than X. Once this point is reached, then the number of moduli being used in the *Approximate CRT Reconstruction* will be equal to the length of X.

At any one stage of the calculation it will be assumed that n' moduli are being used and that $M' = \prod_{i=0}^{n'-1} p_i$. If n is the original number of moduli then the approximation error in any reconstruction will be less than $max\_error = n/2^d$ as $n' \leq n$. If $len(X) < n'$ then $M'/p_{n'-1} > X$, this follows directly from Equation 2.23. Ideally therefore a modulus would be removed if Equation 2.29 holds.

$$\frac{X}{M'} \leq \frac{1}{p_{n'-1}} \tag{2.29}$$

As was discussed in subsection 2.2.5, to ensure that errors do not cause ambiguity between very large and very small numbers it is essential that $0 \leq X < M'(1 - max\_error)$, this would change Equation 2.29 to the following:

$$\frac{X}{M'} < \frac{1 - max\_error}{p_{n'-1}} \tag{2.30}$$

As was also discussed in subsection 2.2.5, if the estimate is greater than $1 - max\_error$ then it is a small not a large number. By adding max_error to the estimate and subtracting 1, the possible error becomes positive instead of negative. This comparison and subtraction are not necessary however as the same effect can be achieved using a modular method as shown below. Note that the estimate is $E/2^d$ as was discussed in subsection 2.2.4 and that $frac(x)$ is the fractional part of x.

$$frac\left(\frac{E}{2^d} + max\_error\right) = frac\left(\frac{E}{2^d} + \frac{n}{2^d}\right) = \frac{(E+n) \bmod 2^d}{2^d} \tag{2.31}$$

Note also that if d is chosen to be the length of a built in integer type, then the modular reduction is a by-product of overflow. As this value is guaranteed not to be too small, then it can be substituted into Equation 2.30 to form the following condition that a modulus may be safely removed.

$$\frac{(E+n) \bmod 2^d}{2^d} < \frac{1 - max\_error}{p_{n'-1}}$$

$$(E+n) \bmod 2^d < \frac{2^d - n}{p_{n'-1}} \tag{2.32}$$

If the value of E obeys Equation 2.32, it is certain that $len(X) < n'$. However, to be certain that $len(X) = n'$ then it must be true that $M/p_{n'-1} < X$ as stated in Equation 2.23. So ideally, it would be known that $len(X) = n'$ if $len(X) \leq n'$ and Equation 2.33 holds.

$$\frac{X}{M'} > \frac{1}{p_{n'-1}} \tag{2.33}$$

Because of errors this cannot be tested exactly, but allowing for errors it is certain that $len(X) \not< n'$ if Equation 2.34 holds.

$$\frac{(E+n) \bmod 2^d}{2^d} > \frac{1}{p_{n'-1}} + \frac{n}{2^d}$$

$$(E+n) \bmod 2^d > \frac{2^d}{p_{n'-1}} + n \tag{2.34}$$

This leaves the case when neither Equation 2.32 or Equation 2.34 holds. If this happens the length is unknown, this condition is shown in Equation 2.35.

$$\frac{2^d - n}{p_{n'-1}} \leq (E+n) \bmod 2^d \leq \frac{2^d}{p_{n'-1}} + n \tag{2.35}$$

If Equation 2.35 holds it is known that $len(X) \leq n'$, while it is clear that some doubt in the length will always remain, it would be preferable to know that $len(X) = n'$ or $len(X) = n' - 1$.

It would be certain that this was the case if the largest number of length $n' - 2$ did not obey Equation 2.35. The maximum value that the estimate can take is the true value so it is certain that Equation 2.36 holds.

$$\frac{E}{2^d} \leq \frac{1}{p_{n'-1}p_{n'-2}}$$

$$(E+n) \bmod 2^d \leq \frac{2^d}{p_{n'-1}p_{n'-2}} + n \tag{2.36}$$

Combining Equation 2.36 with Equation 2.32, it is possible to calculate the required precision needed to ensure the length of a number, calculated using the *Repeated Approximations* method, is at most one too much.

$$\frac{2^d}{p_{n'-1}p_{n'-2}} + n < \frac{2^d - n}{p_{n'-1}}$$

$$2^d > \frac{n(1+p_{n'-1})p_{n'-2}}{p_{n'-2}-1} \tag{2.37}$$

This can be simplified by noticing that $p_{n-1} \leq p_{n'-2} - 1$, this must be true as p_{n-1} is the smallest modulus, and $n' - 2 \leq n - 2$. Similarly as p_0 is the largest modulus $p_0^2 \geq (1 + p_{n'-1})p_{n'-2}$, by making these two substitutions the result is Equation 2.38, which can be applied for any valid value of n'.

$$2^d > \frac{n p_0^2}{p_{n-1}} \tag{2.38}$$

Computational cost of repeated estimates

The cost of each step of the algorithm is dependent on the cost of an *Approximate CRT Reconstruction*. Note that, as the number of moduli decreases, different values of Inv_i will need to be used. If a full set of such constants is stored, the space needed will be $O(n^2)$, as opposed to the $O(n)$ needed for a fixed number of moduli.

The number of steps required will depend on the difference between len(X) and the value of n, and will equal $n - len(X) + 1$. Sequentially the time complexity becomes $O(n(n - len(X)))$ and in parallel it becomes $O((n/nproc + \log(nproc))(n - len(X)))$.

2.3.4 A Probabilistic Binary Search

This method uses three stages to find the length of X. The first stage is a binary search, which uses *Approximate CRT Reconstructions* to find a value for $len(X)$. This value has a high probability of being close to the real value. In the second stage the value for length is given a precise value by using the *Repeated*

Approximations method described in subsection 2.3.3. In the third stage the validity of stage one is tested using a probabilistic confirmation function.

This is a new method for calculating the length of a number in a modular representation. It can also be used for detecting the sign of modular numbers which are stored in the range $-\lceil M/2 \rceil \leq X \leq \lfloor M/2 \rfloor$. An example of this method being used for sign detection is shown in subsection 4.5.4.

Binary Search

To perform the binary search *Approximate CRT Reconstructions* are made using varying numbers of moduli. As was the case in subsection 2.3.3, it is assumed that it is know that $len(X) \leq n$. In each step $n' \leq n$ moduli will be used to perform an approximate reconstruction. As before, the approximation of X/M' will be $E/2^d$, and this will be an underestimate by a maximum of $max\_error < n/2^d$.

For the binary search to work, it will need to be known when an approximation is close to zero. In this case it will be assumed that $len(X) < n'$, if the approximation is not close to zero then it is certain that $len(X) \geq n'$. The condition for an approximation to be assumed close to zero is described by Equation 2.39.

$$(E + n - 1) \bmod 2^d < n \tag{2.39}$$

If the true value of X/M' is $0 \leq X/M' < 1/2^d$, then the approximated value of E will lie in the range $-n < E < 1$, as E is an integer then this becomes $-(n-1) \leq E \leq 0$, hence $(E + n - 1) \bmod 2^d < n$.

Chance of error in binary search

Taking an arbitrary M' with n' moduli such that $M' = p_0 p_1 \ldots p_{n'-1}$ then X, which is known to be less than M, will obey the following equation.

$$X = X' + KM', 0 \leq X' < M' \tag{2.40}$$

If the value of K is zero then setting n' as an upper limit will be correct and no error can occur. However, if K is non-zero then setting n' as an upper limit will be incorrect. Assuming that K is non-zero, the probability that error will occur will depend on the value of X'.

For a random X, in the range $M' < X < M$, with $M' \ll M$, X' can take any value between 0 and $M' - 1$ with equal probability. Likewise assuming $2^d \ll M'$, the value of E approximated can take any value between 0 and $2^d - 1$ with equal probability. As there are n values of E which will result in Equation 2.39 being satisfied, and as E can take any of 2^d values with equal probability, the probability that an error can occur is $n/2^d$.

The binary search will have $\lceil \log_2(n) \rceil$ steps. In the worst case the value of K should be non-zero in every step, this will happen when $len(X) = n - 1$. In this case the probability of an error occurring is shown in Equation 2.41.

$$1 - \left(1 - n2^{-d}\right)^{\lceil \log_2(n) \rceil} \tag{2.41}$$

Getting an exact value for $len(X)$

At the end of the binary search, the upper bound will correspond to a value of n', which gave a value of E which obeys Equation 2.39; the lower bound will correspond to a value of n' which does not give a value of E obeying Equation 2.39. The lower bound is not necessarily equal to $len(X)$ as to be certain of that Equation 2.34 must hold.

To get a near exact value of $len(X)$ the *Repeated Approximations* method can be used. The number of steps needed will depend on the size of the estimate 2^d. If there have been no errors in the binary search then the true value of X/M' must be greater than $1/2^d$, otherwise Equation 2.39 would have been satisfied. In each step of the repeated approximation the value of X/M' is multiplied by $p_{n'-s}$, where s is the step number.

If all the $p_i > 2^t$, then after s steps the value of X/M' will be at least.

$$X/M' \geq 2^{st-d} \tag{2.42}$$

If this value is greater or equal to 1, then the value of s is too much. This value will be greater or equal to 1 if $st \geq d$. Hence the true value of s will obey Equation 2.43.

$$s < d/t \tag{2.43}$$

Confirmation

Once the value of $len(X)$ is thought to be known it can then be checked. If the value of $len(X)$ is correct then the value of K in Equation 2.40 will be equal to zero, otherwise it will have a value between 1 and M/M'. To confirm that the value of $len(X)$ is correct the value of K is reconstructed using a random set of moduli. If $K = 0$ then all these reconstructions will be zero. If it is not then these reconstructions will only be zero if the moduli chosen all divide the value of K.

The value of K can be found by rearranging Equation 2.40.

$$K = \frac{X - X'}{M'} \tag{2.44}$$

This will only be equal to zero mod m if $X \equiv X' \bmod m$. In Section 2.2.6 a method was shown to perform a *Reconstruction to a Small Modulus*. By using differing numbers of moduli in the reconstructions involved, the values of X mod m and X' mod m can be calculated. If these values are equal, then $m|K$.

If s steps are used in which the values of m are all greater than 2^t, and all the m's are relatively prime then the chance that a random K will pass this test is $P(s)$, which obeys Equation 2.45.

$$P(s) < \frac{1}{2^{st}} \tag{2.45}$$

Note that if $0 < K < 2^{st}$, the test will always fail as a non-zero number cannot be divisible by a number which is greater than itself.

What to do when the test is failed

If a number fails the test it is because it is close to a multiple M'. Once this is known it is not easy to alter a number so that the binary search will succeed. Scaling or adding to the number does not help in determining the true length.

The simplest solution would be to use one of the other two length algorithms instead. However, it is possible to use what has been learned so far to our advantage. A minimum length is now known, as the lower bound is never too high. It is also known what the value X mod m is for a value of m. A second binary search can be performed using the value of X' mod m to determine whether the

n' moduli are sufficient to reconstruct the number.

This will lead to a second value for $len(X)$, which can be tested as before using a new set of moduli. By repeating this process the true value of $len(X)$ can be found. Indeed, this method could be used in place of the original binary search, but despite having the same complexity, the *Approximate CRT Reconstruction* is quicker than the *Reconstruction to a Small Modulus* and so is used as the first choice.

Computational cost of Probabilistic Binary Search

When using the *Probabilistic Binary Search* method the first stage is a binary search. The binary search will require $\lceil \log_2(n) \rceil$ steps. In each step a single *Approximate CRT Reconstruction* will be performed. A single *Approximate CRT Reconstruction* has a sequential complexity of $O(n)$ and a parallel complexity of $O((n/nproc + \log(nproc))$. This leads to the binary search stage having a sequential complexity of $O(n \log(n))$ and a parallel complexity of $O((n/nproc + \log(nproc)) \log(n))$.

The number of steps needed to get a precise value of $len(X)$ depends on the relative sizes of the bits of accuracy in the reconstruction and the size of the moduli. This is independent of the size of X, as each step is an *Approximate CRT Reconstruction* it will not add to the complexity overall.

The final conformation step will take a number of *Reconstruction to a Small Modulus* reconstructions depending on the desired certainty. The number of steps needed will not be related to $len(X)$ or the value of n. It is thus taken to be a constant. In subsection 2.2.6, it was shown that the complexity of such a reconstruction is the same as an *Approximate CRT Reconstruction*, and so it will not affect the overall complexity.

The overall complexity is determined by the binary search step and and is thus $O(n \log(n))$ sequentially, and $O((n/nproc + \log(nproc)) \log(n))$ in parallel.

2.3.5 Comparison of length algorithms

We have seen three different methods for calculating the length of a number. These are *Mixed Radix Conversion*, *Repeated Approximation* and *Probabilistic Binary Search.*

The *Mixed Radix Conversion* is good for numbers which have lengths which are significantly smaller than the number of moduli n. This would be useful if a number is calculated to be less than a certain bound, as these bounds are often significantly larger than the real answer.

The *Repeated Approximation* method is good for numbers with lengths close to n. Assuming that the lengths of the inputs to a calculation are known it may well be possible to give an accurate bound on the maximum length of the result. The difference between this maximum length and the real length will be the number of steps required.

If the length of a number is neither likely to be very small or close to a known bound n, then the *Probabilistic Binary Search* is the method with the lowest complexity. It is much more complicated than the other two methods, and will require numbers of a significant length before an advantage is seen.

2.4 Base extension

2.4.1 Definition of base extension

Base extension is increasing the number of moduli used to store a number. Up to this point it has been assumed that a fixed number of moduli would be used for each calculation, but the number of moduli required will depend on the size of the number being represented. Before a base extension begins it is assumed that the number X is less than the product of all moduli M. Base extension must take place before the number is increased in size.

If the old set of moduli (in this context the modulus base), is $\{p_0, p_1, \ldots, p_{n-1}\}$, the new extended set of moduli will be $\{p_0, p_1, \ldots, p_{n-1}, p_n, \ldots, p_{n+x-1}\}$, where x is the number of extra moduli added. As was the case with the original moduli the new moduli are assumed to be relatively prime to both each other and the original moduli.

Three methods are discussed in this section. The first is, *Individual Reconstructions*, which involves the *Reconstruction to a Small Modulus* described in subsection 2.2.6. The second method, *Mixed Radix Extension* uses the *Mixed Radix Conversion* discussed in subsection 2.3.2. The third, *Combined Individual Reconstructions* combines several *Individual Reconstructions* decreasing both

calculation and communication.

2.4.2 Individual Reconstructions

In subsection 2.2.6, a method was described to perform a *Reconstruction to a Small Modulus*, where a small modulus was one that could be handled using a machine integer type. It is assumed that the new moduli $p_n, \ldots, p_{n+x-1}$ are also small, so each modulus can be constructed using this method. This method is a natural extension of the *Reconstruction to a Small Modulus.*

To extend the number base by a single modulus, a single use of the *Reconstruction to a Small Modulus* can be used, but if $x > 1$ then the value of k will only need to be calculated once, as this value is dependent on the values of X and M and not the new moduli. This can be seen in Equation 2.21, which is copied below.

$$X \equiv M \left(\left(\sum_{i=0}^{n-1} p_i^{-1} y_i \right) - k \right) \pmod{m}$$

Computational cost of Individual Reconstructions

The time complexity to calculate x additional moduli will be x times the complexity of a single *Reconstruction to a Small Modulus.* The time saving of not recalculating k does not affect the complexity. In practice it is the calculation of $p_i^{-1} \bmod m$ which is the most time consuming part.

The time complexity for the whole calculation is thus $O(x\,n)$ sequentially and $O((x\,n)/nproc) + O(x\,\log(nproc))$ in parallel.

2.4.3 Mixed Radix Extension

Performing base extensions using a Mixed Radix Extension was well established by the time Szabo and Tanaka wrote their 1967 book *Residue Arithmetic and its Applications to Computer Technology* [37].

The first stage of this method is to perform a *Mixed Radix Conversion* as described in subsection 2.3.2. Once this conversion is complete, the value of X will be known in terms of the constants $d_0, d_1 \ldots d_{n-1}$. Knowing these constants, the value of X can be reconstructed according to Equation 2.46.

$$X = d_0 + d_1 p_0 + d_2 p_0 p_1 + ... + d_{n-1} p_0 p_1 \ldots p_{n-2} \tag{2.46}$$

If it is assumed that all the p_i are known to each processor, then to calculate X mod m is a matter of evaluating Equation 2.46 mod m. This can be most simply achieved by using Horner's Rule as in Equation 2.47.

$$X \equiv d_0 + p_0(d_1 + p_1(d_2 + ... + p_{n-2}(d_{n-1}) \ldots)) \bmod p_i, i \geq n \tag{2.47}$$

Computational cost of Mixed Radix Extension

The cost of the *Mixed Radix Conversion* was shown in subsection 2.3.2. Sequentially it was $O(n\,len(X))$, and in parallel it was $O((n\,len(X))/nproc)$, with a possible extra $O(len(X)\log(nproc))$ for communication.

However, if $len(X)$ is already known then the initial value of n used in the *Mixed Radix Conversion* could be set to $len(X)$. This will reduce the costs to $O(len(X)^2)$ sequentially and to $O(len(X)^2/nproc) + O(len(X)\log(nproc))$ in parallel.

The cost of calculating the values for each new modulus will be $O(len(X))$, and so for x new moduli the sequential time is $O(x\,len(X))$. In parallel each processor will need to deal with up to $\lceil x/nproc \rceil$ moduli, giving a complexity of $O((x\,len(X))/nproc)$. Note that, no communication will be needed after the *Mixed Radix Conversion* has been completed.

If it is assumed that $len(X)$ is already known, the total complexities for the *Mixed Radix Extension* are sequentially $O(x\,len(X)) + O(len(X)^2)$ and in parallel $O((x\,len(X))/nproc) + O(len(X)^2/nproc)$, with a possible $O(len(X)\log(nproc))$ extra for communication.

Reducing the communication costs

As discussed in subsection 2.3.2, it is possible to perform broadcasts and globalor operations in constant time using relatively simple hardware. If, however, this is not available there is an additional communication cost associated with each step of a *Mixed Radix Extension.*

In each step two communications are made: a broadcast of the previously

calculated value d_i, and a globalor to check that the value remaining is not zero. If $len(X)$ is known, then the globalor will not be required; if it is not known, then it is safe to set it to n as that is an upper limit for the length. Calculating a mixed radix conversion past the length of a number will result in the calculation of redundant values d_i, all of which will be zero. It will not affect the result however, so it is safe, if wasteful.

The values of d_i still need to be broadcast and $len(X)$ broadcasts of a single integer are made using the original method. It would be better to make a single broadcast of $len(X)$ integers if a message passing system was being used. This is because there is a latency inherent in every message that is passed, and often also a minimum message size.

It is not possible to delay all the messages to the end of the calculation, but it is possible to reorder the calculation to reduce the number of messages. If there are *nproc* processors it is assumed that each processor has an index *iproc* which runs from 0 to $nproc-1$. Each processor will store x_i only if $i \equiv iproc \bmod nproc$. If this is the case the mixed radix conversion can be reordered so that instead of reducing in the order $p_0, p_1, \ldots, p_{n-1}$, the moduli on processor 0 are considered first, then those on processor 1, and so forth. Hence, if 4 processors are used and $n = 8$, the new order would be $p_0, p_4, p_1, p_5, p_2, p_6, p_3, p_7$. This would not be satisfactory if length was being calculated, as the order of moduli is significant for length, but it will not affect the validity of the conversion for the purpose of base extension.

Once the reordering has been made, it is possible for processor 0, to calculate all the first $n/nproc$ values of d_i without communicating with the other processors. It can then broadcast all the values of d_i, which the other processors can use to reduce their values of x_i. Processor 1 can then calculate the next block of values of d_i, and the process is repeated. When the last processor has finished broadcasting, all the values of d_i are known to all the processors. The calculation can then proceed as before taking account of the new order of moduli.

The total number of communications will be *nproc* each of size $n/nproc$ integers. There will be a time when only one processor can be performing calculations, but the wasted time will be less than 1 in *nproc* assuming asynchronous operation.

2.4.4 Combined Individual Reconstructions

The *Combined Individual Reconstructions* presented below is based on the base extension circuit proposed in [31].

When performing multiple individual reconstructions a parallel reduction is used for each extra modulus. In *Combined Individual Reconstructions* this is not the case. Instead a number of sequential *Reconstructions to a small modulus* are performed. As it is assumed that the values of x_i are distributed among the processors, the reconstructions cannot take place without prior communication. If we again consider Equation 2.21, which is copied below, we can see that it is the values of y_i and k which are needed.

$$X \equiv M\left(\left(\sum_{i=0}^{n-1} p_i^{-1} y_i\right) - k\right) \pmod{m}$$

If the value of k is not already known, it can be calculated. Then all of the values of y_i can be calculated using locally stored tables as in an approximate reconstruction. All of the values are y_i are then broadcast to each processor. Once the processors know all the values of y_i they can then perform the inner summation using either pre-stored values of $p_i^{-1} \bmod p_j$, or by using the extended Euclidean algorithm. $M \bmod p_j$ could also be stored in tables which would be of size $O(n^2)$.

Computational cost of Combined Individual Reconstructions

Calculating all the values of y_i will require time of $O(n/nproc)$. To broadcast them will require $nproc$ broadcasts of data of size $O(n/nproc)$. Using a binary tree this will involve $O(\log nproc)$ stages, so the total communication cost will be $O(n \log nproc)$. If the length of the number is already known, n can be replaced with $len(X)$.

To calculate each new residue will take time of $O(n)$, as there are x of these split between $nproc$ processors this will take time equal to $O(nx/nproc)$.

The overall complexity is thus $O(nx/nproc)$ for calculation and $O(n \log nproc)$ for communication.

2.4.5 Comparison of methods

Sequential

In the sequential case, the time complexity of *Individual Reconstructions* $O(x\,n)$, is the same as *Combined Individual Reconstructions*, and can only be worse than *Mixed Radix Extension* at $O(x\,len(X)) + O(len(X)^2)$ if the length is not known and x is large. In practice, in tests where the residue base is doubled, *Individual Reconstructions* is slower than *Mixed Radix Extension*, which in turn is slower than *Combined Individual Reconstructions.*

If the size of x is small, then *Combined Individual Reconstructions* is still likely to be quicker than *Individual Reconstructions* apart from the trivial case of $x = 1$.

Parallel

In the parallel case, the communication time of *Individual Reconstructions*, which is $O(x\ \log(nproc))$, will be smaller than those for either *Mixed Radix Extension*, which is $O(len(X)\log(nproc))$, or *Combined Individual Reconstructions*, which is $O(n \log nproc)$, when x is much smaller than the length of the number. However, the advantage is not so significant as the communications of both *Mixed Radix Extension* and *Combined Individual Reconstructions* are in $nproc$ stages so when $nproc$ is significantly less than x the advantage will be less.

In theory, when x is large, there should be no difference between *Mixed Radix Extension* and *Combined Individual Reconstructions.* However, in tests *Combined Individual Reconstructions* was again faster in parallel when the residue base is doubled, and the length of the number was known.

2.5 Conclusions

In this chapter three problems have been discussed. The first is the problem of gaining information about a number stored in a modular representation without the cost of a full *Chinese Remainder Theorem* reconstruction.

In an *Approximate CRT reconstruction* an estimate of X/M is formed. The estimate is built up using integer fractions, replacing the floating point estimates which had previously been proposed. In a *Reconstruction to a small modulus* a

new algorithm used to perform a modular reduction. Both of these algorithms can be performed with a sequential time complexity of $O(n)$.

The second problem was to determine the length of a number stored in a modular representation. A new definition of number length was presented which is specific to the modular representation.

Three methods of calculating length were shown, two of which used the techniques developed in Section 2.2, and one of which was based on a *Mixed Radix Conversion.* Both the *Repeated Approximations* and the *Probabilistic Binary Search* are new algorithms based on the methods developed in Section 2.2. It was shown that the worst case, using the two deterministic methods was $O(n^2)$ sequentially but by using a probabilistic method this could be reduced to $O(n \log(n))$.

The third problem was to increase the number of residues used to store a number. It was shown that for small increases in the number of residues, it was best to use the *Reconstruction to a Small Modulus* described in Section 2.2. For larger increases in the number of residues, *Mixed Radix Extension* and *Combined Individual Reconstructions* both give better results.

Throughout the chapter the importance of parallel reduction operations have been discussed, and it has been shown that these will determine the efficiency of any parallel implementation.

Part II

A library for parallel modular arithmetic

Chapter 3

Arithmetic using approximations

3.1 Introduction

In this chapter is a description of a library of functions, that allow arithmetic to be performed on bignums using a modular representation, which is hidden from the end user. As well as storing the residues of the number, the length of the number (Section 2.3) is stored with an approximation of the number, reconstructed to this length.

The number of moduli used to store a number will vary according to its length. When extra residues are required a base extension is performed. To be able to use the approximations for comparison, it is important that the error is bounded, so all approximations are maintained at the same level of accuracy.

The residues are distributed in a data parallel manner, which again, is hidden from the end user, who is able to write sequential code. It is thus possible to write a single piece of source code, and compile for a range of platforms using a range of different communication mechanisms.

3.2 The choice of moduli

There are two versions of the library being described in this chapter. The first uses 16 bit moduli, and the second uses 32 bit moduli. The main section of this chapter will deal with 16 bit moduli. The modifications needed to cope with 32 bit moduli are discussed in Section 3.13.

Related to the size of the moduli is the size of the approximations used in the approximate CRT reconstructions. When 16 bit moduli are used the approximations are made with 32 bits. When 32 bit moduli are used the approximations are made with 64 bits.

One method of choosing 16 bit moduli would be to start at $2^{16} - 1$, and to work downward, selecting a number if it is relatively prime to all those currently chosen. Hence, the first five moduli would become:

$$p_0 = 65535, p_1 = 65534, p_2 = 65533, p_3 = 65531, p_4 = 65521. \tag{3.1}$$

This appears to be much better than the largest five prime moduli less than 2^{16}, which are:

$$p_0 = 65521, p_1 = 65519, p_2 = 65497, p_3 = 65479, p_4 = 65449. \tag{3.2}$$

However, as the number of moduli increases all the smaller prime numbers become used up, and the advantage of using composite moduli diminishes. Indeed just in the first four composite moduli the list of factors includes 2,3,5,7,13,17,19 and 31.

The disadvantage of using composite moduli is in the calculation of inverses. With a prime modulus, an inverse will always exist if the residue is not zero. With a composite modulus, such as 65535, divisibility by 3,5,17 and 257 would have to be checked before an inverse could be calculated. It is for this reason that only prime moduli are used: hence the use of p_i (not m_i) throughout this thesis.

To gain maximum efficiency the prime moduli are in strictly decreasing order. This was assumed in Section 2.3 which dealt with number length.

Once the type of moduli has been chosen, then the number of moduli needs to be considered. There are 6542 primes less than 65536, but to simplify calculations only 1024 primes are used. This makes it possible to bound the errors in the approximate reconstruction as is discussed in Section 3.6. It also reduces the problems associated with calculating length when multiplying, which are discussed in Section 3.11.

3.3 Datatype

The datatype for numbers stored with approximations is called `t_PMA`. This is defined as a pointer to `struct T_PMA` which can be seen in Table 3.1. The number of residues used to store each number is `num_res`, which corresponds to the size of the array, which is accessed through the pointer `array`. Negative numbers are handled by the field `neg`, which is true if the number is negative.

The length of the number is stored in the field `length`. An approximate reconstruction is performed using `length` moduli and is stored in `approx`. The approximation is stored in a 32 bit integer which should be treated as a binary fraction. Finally the value of k, as described in subsection 2.2.5, is stored in the field k.

To create a number the function `create` is called, and to destroy it the function `destroy` is called. The allocation of the memory for the array of residues is performed automatically, as and when it is needed.

Field	Type	MPL Type
neg	`int`	singular
array	`unsigned int*`	plural
num_res	`int`	singular
length	`int`	singular
approx	`unsigned int`	singular
k	`int`	singular

Table 3.1: Fields of structure T_PMA

In Table 3.1, is a column labelled *MPL Type.* This refers to where the data is stored.

On the Maspar MP1/2, which is described in Section 5.2, data can either be stored on the ACU (Array Control Unit) or on the individual processors. Data which is stored on the ACU is called singular data and data which is stored on the individual processors is called plural data. As can be seen in the table, only the array of residues is stored on each of the processors.

On a distributed computer there is no such central processor to store the singular data, on these computers the singular data is duplicated on each of the processors. To ensure that this data is consistent, it is important that each processor performs the same calculations using the same data. By using broadcasts

and reductions in the parallel segments of calculations it can be certain that each of the processors have the same data.

The part of each number which is not stored on each of the processors is the array of residues. This is divided evenly. The value of `num_res` is always the number of processors multiplied by a power of 2. By using a power of 2 the smallest base extension will involve doubling the number of residues. Performing a few large base extensions is more efficient than performing many small ones.

The number of processors *nproc* and the processor index *iproc* are stored in the variables `NPROC` and `IPROC`. The processor with index `IPROC` handles all of the moduli p_i where $i \equiv$ `IPROC` mod `NPROC`.

3.4 Tables of pre-calculated data

3.4.1 The moduli

There are `No_Primes16` moduli, which are stored in an array of `unsigned short` called `Primes16`. This array is distributed between the processors, as was described in Section 3.2, so that each processor stores only a fraction of the moduli.

While most calculations can be performed without a processor needing access to the other moduli, storing a full set of moduli can reduce the number of broadcast operations in both general and exact division. Where it is appropriate a full set of moduli are stored in the array `All_Primes16`.

With 1024 moduli stored as `unsigned short`, a total of 2 kilobytes of memory will be required to store all the moduli.

3.4.2 Approx Reconstruction constants

To perform an approximate reconstruction the value of Inv_i is needed, where $Inv_i = (M/p_i)^{-1} \bmod p_i$, the value of M will vary according to n the number of moduli being used for the reconstruction, and the value of i will range between 0 and $n-1$. These values are stored in the two dimensional array `Inverses16`.

The array is indexed so that the value of Inv_i using n moduli will be stored on processor i mod `NPROC`, in `Inverses16[i/NPROC][n]`. Using 1024 moduli and storing the inverses as `unsigned short`, the total memory used will be 2 megabytes, which is split evenly between the processors.

To speed up the calculations in the approximate reconstruction it is also useful to store $2^{32} \bmod p_i$, and $p_i^{-1} \bmod 2^{32}$. These are stored in the arrays `two_32` and `p_inv`. They are both stored as `unsigned int` and take up 4 kilobytes of memory each. Again these are split between the processors.

3.4.3 Modular Reconstruction constants

The equation, for calculating a modular reconstruction to an arbitrary modulus m, is given in Equation 2.21, and is duplicated below.

$$X \equiv M \left(\left(\sum_{i=0}^{n-1} p_i^{-1} y_i \right) - k \right) \pmod{m}$$

Assuming that the values of y_i and k are known, it will be necessary to calculate $p_i^{-1} \bmod m$ and $M \bmod m$. The most common form of modular reconstruction is during base extension, when m will be one of the moduli p_i.

While both of these calculations can be performed without affecting the time complexity of the calculation (see subsection 2.2.6), as base extension is a very common operation, two extra tables are stored.

To store $M \bmod p_i$ for all values of n would involve the same size table as `Inverses16`. However, during base extension, $M \bmod p_i$ is only needed when $i > n$. Inv_i is only defined when $i \leq n$. Hence, it is possible to put both sets of values in the same table without affecting its size.

To store $p_j^{-1} \bmod p_i$ for all $i, j < n$, except when $i = j$, is another 2 megabytes of data. While it is again true that only half of the values are needed for base extension, that is when $j < i$, the other values can be of use in exact division where it may be necessary to divide by one of the moduli. The values are stored in the array `p_inverse`.

One modular reconstruction that can be performed extremely rapidly is when $m = 2^{32}$. This is particularly fast as arithmetic modulo 2^{32} is a side effect of normal integer arithmetic. The value of $p_i^{-1} \bmod 2^{32}$ is stored in `p_inv` which was needed for the approximate reconstruction, the value of $M \bmod 2^{32}$ is stored in `M_32`, However, as this array is indexed on n, it cannot be split between the processors like the other tables. Hence each processor needs to store an extra 4 kilobytes.

3.4.4 Total size of tables

All the tables are shown in Table 3.2, along with their size, when 1024 moduli are used. On a distributed computer, singular tables are stored on each processor in full, plural tables are shared between them.

Field	Size (Bytes)	MPL Type
`Inverses16`	2,097,152	plural
`p_inverse`	2,097,152	plural
`Primes16`	2,048	plural
`two_32`	4,096	plural
`p_inv`	4,096	plural
`All_Primes16`	2,048	singular
`M_32`	4,096	singular
`total plural`	4,200,544	plural
`total singular`	6,144	singular

Table 3.2: Size of tables using 1024 16-bit moduli

For example, if the number of processors was 4, each processor would need to store $4200544/4 + 6144 = 1056280$ bytes.

3.5 Initialisation

Before the functions in the library can be used the program must call the function **`initialise`**. This function performs two main tasks: it ensures that the tables of data are stored in memory, and that the parallel processes are launched and synchronised.

The tables are loaded from a data file **`primes_data`**, which is stored in a location determined by the main header file **`PMA.h`**. With a large number of processors it may be quicker to calculate the tables, but this file contains at least the list of moduli which is calculated using the primality testing function of the Gnu Multiple Precision library.

Parallel communication is performed using two different libraries of functions called MPI and AFAPI. Details of these communication libraries can be found in Section5.2.

Using both MPI and AFAPI, it is necessary to call an initialisation function to ensure that each process is communicating properly with the others. It also, in the case of MPI, ensures that the command line arguments are passed correctly. By including this in the initialisation function, these details can be hidden from the user.

To free up the memory taken by the tables, and to shut down the parallel processes, the function `finalise` is called. Again this hides the parallel aspects from the user, calling the appropriate functions from the communication libraries.

3.6 Approximate Reconstructions

3.6.1 approxCRT

Definition

The function `approxCRT` takes as its arguments a number of type `t_PMA` and the number of residues to be used in the reconstruction. It returns an unsigned integer which is the approximation E. The value of E should obey Equation 3.3, where $0 \leq error < 2^{-22}$. $(1024/2^{32} = 2^{-22})$

$$\frac{E}{2^{32}} = frac\left(\frac{X}{M} - error\right) \tag{3.3}$$

Calculation

The value of E is the sum of all E_i reduced modulo 2^{32}.

$$E = \left|\sum_{i=0}^{n-1} E_i\right|_{2^{32}} \tag{3.4}$$

The method used to calculate E_i was described in subsection 2.2.4. It assumes the value m_i the error term is already known.

$$E_i = \left|-m_i(p_i)^{-1}\right|_{2^{32}} \tag{3.5}$$

The value of m_i was defined in subsection 2.2.3; it is dependent on the value of y_i.

$$m_i = |y_i 2^{32}|_{p_i} \tag{3.6}$$

The value of y_i is the CRT reconstruction term, which was defined in subsection 2.2.1.

$$y_i = |Inv_i x_i|_{p_i} \tag{3.7}$$

It is now possible to perform the calculation by first calculating y_i, then m_i and finally E_i. To calculate y_i requires x_i, which is stored in the field `array`, and Inv_i, which is stored in `Inverses16`. Both of these are of size less than p_i, which is less than 2^{16}, and hence, can be multiplied together using an `unsigned int` before being reduced modulo p_i.

To calculate m_i, the value of $2^{32} \bmod p_i$ is taken from the array `two_32` and multiplied by y_i. Again this product fits into an `unsigned int`, and is reduced modulo p_i.

To calculate E_i, the value of m_i is multiplied by the inverse of p_i which is stored in the array `p_inv`. It is not necessary to try and reduce this answer as that is done automatically. The value is then made negative, before being added to an accumulated sum.

When each processor has finished summing all the values of E_i for which it is responsible, a parallel reduction is performed with natural overflow taking care of the reduction modulo 2^{32}.

Correctness

To see that the above calculation does indeed produce an estimate of X/M, and further, to see that the estimate is accurate to within 2^{-22}, it is necessary to use the equations built up in Section 2.2.

In the previous subsection it has been seen that E is the sum of the values E_i.

$$E = \left| \sum_{i=0}^{n-1} E_i \right|_{2^{32}} \tag{3.8}$$

Each of these are equal to the value of y_i/p_i multiplied by 2^{32} and truncated.

$$E_i = \left\lfloor \frac{y_i 2^{32}}{p_i} \right\rfloor = 2^{32}\left(\frac{y_i}{p_i} - e_i\right), 0 \leq e_i < 2^{-32} \tag{3.9}$$

Hence $E_i/2^{32}$ is a good estimate of y_i/p_i. Summing all these E_i leads to the following:

$$E = 2^{32} frac\left(\sum_{i=0}^{n-1} \frac{y_i}{p_i} - \sum_{i=0}^{n-1} e_i\right) \tag{3.10}$$

As the value of n can be, at most $2^{10} = 1024$, then the error term can be, at most $2^{10}2^{-32} = 2^{-22}$. Substituting the term *error* for the total accumulated error gives Equation 3.11.

$$E = 2^{32} frac\left(\sum_{i=0}^{n-1} \frac{y_i}{p_i} - error\right), 0 \leq error < 2^{-22} \tag{3.11}$$

The sum of all the terms y_i/p_i gives $X/M + k$, where k is an integer.

$$E = 2^{32} frac\left(\frac{X}{M} + k - error\right) \tag{3.12}$$

As only the fractional part is wanted k can be removed leading to Equation 3.3 the stated output of `approxCRT`.

$$\frac{E}{2^{32}} = frac\left(\frac{X}{M} - error\right) \tag{3.13}$$

3.6.2 k_approxCRT

Definition

The function `k_approxCRT`, takes as its arguments, a number of type `t_PMA` and the number of residues to be used in the reconstruction, it returns an integer, which is the value k defined as $\lfloor Crt(X)/M \rfloor$. It is assumed that $0 \leq X/M < 1 - 2^{-12}$, else the value of k returned could be equal to $k + 1$.

Calculation

To calculate the value of k, the same calculation is performed as in `approxCRT`. However, after calculating each of the values E_i, they are bit shifted to the right

by 10 bits. The bit shifted values of E_i are then summed as before using a parallel reduction.

The value of k will accumulate in the ten most significant bits. To get this value, the sum is bit shifted right by 22 bits. The other 22 bits are also needed though, as it could be possible that due to errors in the bit shifted values of E_i, a very small number may have a value of k which is 1 too small. To check this, the masked off 22 least significant bits are compared with $2^{22} - 2^{10}$. If they are greater than this number, then 1 is added to the returned value of k.

Correctness

If the bit shifted values of E_i are called F_i, then they will be equivalent to a 22 bit approximation of y_i/p_i.

$$F_i = \left\lfloor \frac{E_i}{2^{10}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{y_i 2^{32}}{p_i} \right\rfloor}{2^{10}} \right\rfloor = \left\lfloor \frac{y_i 2^{22}}{p_i} \right\rfloor = 2^{22} \left(\frac{y_i}{p_i} - e_i \right), 0 \leq e_i < 2^{-22} \tag{3.14}$$

When these are summed this will become Equation 3.15.

$$\sum_{i=0}^{n-1} F_i = 2^{22} \left(\sum_{i=0}^{n-1} \frac{y_i}{p_i} - error \right), 0 \leq error < 2^{-12} \tag{3.15}$$

Again as the sum of all y_i/p_i is equal to $X/M + k$ then this simplifies to the following:

$$\sum_{i=0}^{n-1} F_i = 2^{22} \left(X/M + k - error \right) = 2^{22} k + 2^{22} \left(X/M - error \right) \tag{3.16}$$

As long as $0 \leq (X/M - error) < 1$, then dividing the sum of all F_i by 2^{22} will give the correct value of k. However, it is possible that the error is greater than X/M giving an incorrect answer.

As $0 \leq X/M < 1$ and $0 \leq error < 2^{-12}$ then $-2^{10} < 2^{22}(X/M - error) < 2^{22}$. Any remainder greater than $2^{22} - 2^{10}$ could potentially represent a negative value of $X/M - error$, and hence 1 is added to k in this case.

To ensure that 1 is not added when X/M is large, the range of X/M is restricted to $0 \leq X/M < 1 - 2^{-12}$, which restricts the value of $X/M - error$ to $-2^{-12} < (X/M - error) < 1 - 2^{-12}$.

Increasing the value of n

Increasing the value of n will have a two fold effect on the precision of the estimate of X/M. When $n = 2^{10}$ and $d = 32$ as above, the estimate of X/M was only accurate to $32 - (2 * 10) = 12$ bits. This required the range of X/M to be restricted to $0 \leq X/M < 1 - 2^{-12}$.

In general the estimate of X/M will be accurate to $d - 2\lceil \log_2 n \rceil$ bits, limiting the range of X/M to $0 \leq X/M < 1 - 2^{-(d-2\lceil \log_2 n \rceil)}$.

If the value of n where to be increased the function `establish_length` would also need to be altered so that the values of X/M lie in the correct range.

3.6.3 `t_approxCRT`

Definition

The function `t_approxCRT` takes, as its arguments, a number of type `t_PMA` and the number of residues to be used in the reconstruction. It returns an unsigned integer, which is the approximation $\hat{E}$. The value of $\hat{E}$ should obey Equation 3.17,

$$\frac{\hat{E}}{2^{32}} = \frac{X}{M} \pm \epsilon \tag{3.17}$$

where $0 \leq \epsilon < 2^{-32}$, and $0 \leq \hat{E} < 2^{32}$, assuming that $0 \leq X/M < 1 - 2^{-48}$.

Calculation

In subsection 2.2.4, a method was discussed which could extend a 32 bit approximation of y_i/p_i into a 64 bit approximation. The method is to take the error term m_i, and to repeat the approximation calculation on m_i/p_i. The result is the following approximation of y_i/p_i, where $E_i = \lfloor y_i 2^{32}/p_i \rfloor$, $m_i = |2^{32} y_i|_{p_i}$ and $E_i' = \lfloor m_i 2^{32}/p_i \rfloor$.

$$\frac{y_i}{p_i} = \frac{E_i}{2^{32}} + \frac{E_i'}{2^{64}} + \frac{m_i'}{p_i 2^{64}}, 0 < m_i' < p_i \tag{3.18}$$

To increase the accuracy of E, which is returned by `approxCRT`, the values of E_i' are used. However, as the returned value is only 32 bits long, only the overflow of the sum of all E_i' is needed. That overflow is returned by `k_approxCRT`.

That actual calculation takes place in four stages.

- Use `approxCRT` to calculate E
- Set $x_i = |x_i 2^{32}|_{p_i}$
- Use `k_approxCRT` on scaled x_i to calculate overflow
- Return E plus overflow $\bmod 2^{32}$

Correctness

The value of E returned by `approxCRT` obeys Equation 3.3. The exact value of the error term is,

$$error = \sum_{i=0}^{n-1} \frac{m_i}{2^{32} p_i} = \frac{1}{2^{32}} \sum_{i=0}^{n-1} \frac{m_i}{p_i}. \tag{3.19}$$

As $CRT(X)/M = \sum_{i=0}^{n-1} \frac{y_i}{p_i}$, and $m_i = |2^{32} y_i|_{p_i}$ then the error can be rewritten as,

$$error = \frac{1}{2^{32}} CRT(|2^{32}X|_M)/M = \frac{1}{2^{32}} \left(k' + \frac{|2^{32}X|_M}{M} \right). \tag{3.20}$$

As a 32 bit estimate is returned by `t_approxCRT`, only the value of k' is needed. By adding k' to E the resulting error would become.

$$new\_error = error - \frac{k'}{2^{32}} = \frac{|2^{32}X|_M}{2^{32}M} \tag{3.21}$$

$$0 \leq new\_error < \frac{1}{2^{32}} \tag{3.22}$$

Unfortunately the function `k_approxCRT` will only return the correct value of k' if $0 \leq |2^{32}X|_M/M < 1 - 2^{-12}$ else the value of k' returned could be equal to $k'+1$. As no restriction is placed on $|2^{32}X|_M$ the real error could be $new\_error - 1/2^{32}$, the real error is thus bounded by Equation 3.23. (Note it is not possible to have $real\_error = -\frac{1}{2^{32}}$, as new_error will only be equal to zero if $X = 0$).

$$-\frac{1}{2^{32}} < real\_error < \frac{1}{2^{32}} \tag{3.23}$$

It was also stated that $0 \leq \hat{E} < 2^{32}$, assuming that $0 \leq X/M < 1 - 2^{-48}$. That is to say that a negative error will never force $\hat{E}$ to become negative however small the value of X is relative to M, and that a positive error will never force $\hat{E}$ to become greater than $2^{32} - 1$.

A negative error is a natural by-product of truncation caused by `k_approxCRT`. As this is a truncation, it cannot force a number which is greater than zero to become negative.

A positive error can be caused by passing a number to `k_approxCRT`, which is greater than $1 - 2^{-12}$. This will only cause $\hat{E}$ to become too large if $X/M > 1 - 2^{-32}$ and $|2^{32}X|_M > 1 - 2^{-12}$. This would require X/M to be greater than $1 - 2^{-48}$.

3.6.4 establish_length

Definition

The function `establish_length` takes, as its arguments, a number of type `t_PMA` in which the `length` field has been set to an integer, which is at least as long as the numbers length. It modifies the number, which it is given, so that the `length` field is set to the numbers length, and that the `k` and `approx` fields are set using `k_approxCRT` and `t_approxCRT`.

The length of a number stored in a modular representation was defined in Section 2.3. It is the least number of residues needed to reconstruct the number correctly. Several methods where described in Section 2.3, but the one that is being used is the *Repeated approximation* method described in subsection 2.3.3.

Repeated approximation may overestimate the length of a number, due to ambiguity caused by errors in the approximation. It was shown that this overestimation would be restricted to at most one, if Equation 2.38 held. This is copied below.

$$2^d > \frac{n\,p_0^2}{p_{n-1}}$$

Assuming `approxCRT` is being used the value of d will be 32, n is assumed to

be 1024, $p_0 = 65521$ and $p_{1023} = 54251$. The condition of Equation 2.38 is easily met as can be seen below.

$$2^{32} = 4294967296 > \frac{1024\ 65521^2}{54251} > 81031381 \tag{3.24}$$

The length returned will also need to ensure that $0 \leq X/M < 1 - 2^{-12}$, where M is the product of the first length moduli. This will ensure that `k_approxCRT` returns the correct result. This will also ensure that, the output of `t_approxCRT` is between 0 and $2^{32} - 1$.

Calculation

To establish the length of a number, the function `approxCRT` is called repeatedly, using a diminishing numbers of moduli. If the estimate is small enough, it will be safe to remove another moduli. Otherwise, the length has been found.

Instead of following the method described in subsection 2.3.3 exactly, a fixed bound is used to determine if an estimate is small enough for another moduli to be removed.

The value E returned by `approxCRT` has 1024 added to it to ensure that it is both positive, and that it is an overestimate. This cannot cause a large number to overflow as $0 \leq X/M < 1 - 2^{-12}$, which is still less than 1 when $1024/2^{32} = 2^{-22}$ is added to it.

A modulus will be removed if this number is less than 65536, otherwise it is assumed that the correct length has been found. The number of approximations needed will depend on the accuracy of the value of length set before calling the function.

Correctness

The two criteria which must be met are that the length returned is at most one too much, and that $0 \leq X/M < 1 - 2^{-12}$.

To prove that the first criteria is met, a number whose value of E equals $65536 - 1024$ will be taken. This is the smallest value of E which will not result in another moduli being removed. The true value of X/M will lie between $E/2^{32}$ and $(E + 1024)/2^{32}$, so $X/M \geq 64532/2^{32}$.

If the estimate of the length was at least 2 too much, then it would be possible to multiply X/M by $p_{n-1}p_{n-2}$ and it would still be less than 1. This would not be possible with $X/M \geq 64532/2^{32}$. As the smallest value of p_i is 54251, when the square of this is multiplied by the smallest possible value of X/M, the result is 44221.2, which is significantly larger than 1.

To show that $0 \leq X/M < 1 - 2^{-12}$, the largest value of X/M which has a modulus removed, will be of interest. The largest value of E that will result in the removal of a modulus is $E = 65535 - 1024$. As was previously stated the true value of X/M is bound by $E/2^{32} \leq X/M < (E + 1024)/2^{32}$. Hence the largest possible value of X/M is less than $65535/2^{32}$. When the modulus p_{n-1} is removed the new value of X/M is multiplied by p_{n-1}. The largest value this can take is when the length is reduced from 2 to 1. In this case $p_{n-1} = p_1 = 65519$. Multiplying the old maximum of X/M by this gives the result $X/M < (65519 * 65535)/2^{32} < 1 - 1179631/2^{32} < 1 - 1048576/2^{32} = 1 - 2^{-12}$.

3.6.5 modCRT_32u

Definition

The function `modCRT_32u`, takes as arguments, a number X of type `t_PMA` and a modulus m of type `unsigned int`. It returns X mod m.

Calculation

The method used for this calculation is described in subsection 2.2.6, *Reconstruction to a Small Modulus.* The calculation is summed up in Equation 2.21, which is copied below.

$$X \equiv M \left(\left(\sum_{i=0}^{n-1} p_i^{-1} y_i \right) - k \right) \pmod{m}$$

The values of y_i are calculated using the array **Inverses16**, as was the case in **approxCRT**. The values of p_i^{-1} mod m cannot be taken from tables, however, as there are too many possible values of m. Instead an extended Euclidean algorithm is used.

On each of the platforms tested a 64 bit `unsigned long long` datatype was

available. Using this datatype the values $p_i^{-1}y_i \bmod m$ are calculated. These are then summed, using a parallel reduction, with a 64 bit addition as the operator. The value of k is subtracted from this total. This will have been calculated previously and placed in the field `k`.

The last step is to multiply by $M \bmod m$. A parallel reduction is used to calculate $M \bmod m$ where the operator is multiplication followed by reduction by m.

The use of 64 bit integers could have been avoided by restricting the size of the modulus to 16 bits. The calculation could then be performed using only 32 bit integers. This function is however used by the division function `r_div` which uses 32 bit moduli.

Correctness

The correctness of this function is dependent on the correctness of the value `k`, which is returned by `k_approxCRT`.

3.7 Comparison

3.7.1 Introduction

In subsection 1.1.3 several classical algorithms were described. These included Algorithm 3 which compared two positive numbers. It first compared the lengths of the two numbers, and then, if the lengths are equal, it compared individual digits from the most to the least significant. Only if all the digits are equal will the two numbers be declared equal.

If signed numbers are to be allowed, then the signs of the digits would be compared first before using Algorithm 3. The order of comparison is then as follows:

- Sign
- Length
- Individual digits

Using a modular representation it is not possible to compare individual digits. However an approximation of the value X/M is available, as is the sign and the length of the number. Using these three details of the number, it is possible to return the value of the comparison in the majority of cases.

If the numbers are of the same sign, length, and have approximations which are within error bounds of each other, then two final tests are made. If the approximations are identical, then it is possible that the numbers are equal. So each of the residues is compared. If all the residues are equal, then the numbers must be the same.

Otherwise a new set of residues is calculated, which is the difference between the two numbers. By approximating the size of this number compared with M, a positive number would produce an estimate less than a half, and a negative number would produce a number greater than a half. If the estimate is too small, however, it will not be possible to tell which is the case, and the number of moduli used in the approximation is reduced. Eventually the approximation will be large enough to be able to determine the sign, also giving the length of the difference as a useful by-product.

In summary, the order of comparison using the modular representation is as follows:

- Sign
- Length
- Approximations
- Equality
- Sign of difference

3.7.2 Coping with an overestimate of number length

Detecting a possible overestimate

The function `establish_length` does not always give the correct value of a numbers length. Instead, it sometimes returns a length which is one too much. It is possible that given two numbers X and Y, that $X < Y$ but `establish_length` has set the length of X to be one greater than the length of Y.

To determine if this could have happened we will first establish an upper limit on the value of Y. The function **establish_length** will only remove a modulus if the value of the approximation returned by **approxCRT** is less than or equal to $65535 - 1024$.

The maximum error in the approximation returned by **approxCRT** is less than 1024. Putting these together it is certain that Y is bounded by Equation 3.25, where $M(x)$ is the product of the first x moduli.

$$\frac{Y}{M(len(Y)+1)} \leq \frac{65535}{2^{32}} \tag{3.25}$$

We are only interested in the case when the length of X which was returned by **establish_length** is one more than the length returned for Y. If it can be shown that the value of X is greater than the largest possible value of Y then we can be certain that $X > Y$ and the comparison is finished. If the value of X is less than or equal to the maximum value of Y then it must be possible to reduce the length of X by one.

An approximation of X is stored in the **approx** field. This is calculated using the function **t_approxCRT** which has an error which is less than plus or minus one. If the value returned by **t_approxCRT** is referred to as $\hat{E}$ then the value of X is bounded by Equation 3.26.

$$\frac{X}{M(len(Y)+1)} > \frac{\hat{E}-1}{2^{32}} \tag{3.26}$$

Putting Equation 3.25 and Equation 3.26 together it is certain that $X > Y$ if $\hat{E} \geq 65536$. If $\hat{E} \leq 65535$ then it is possible that the length of X has been overestimated.

If it is possible that the length of X has been overestimated then the length of X can be decreased. The function **establish_length** can then be called to give new values to the **approx** and **k** fields.

For the function **establish_length** to work correctly Equation 3.27 must hold.

$$\frac{X}{M(len(Y))} < 1 - 2^{-12} \tag{3.27}$$

As the function **t_approxCRT** was used to calculate $\hat{E}$ it is certain that Equa-

tion 3.28 holds.

$$\frac{X}{M(len(Y)+1)} < \frac{\hat{E}+1}{2^{32}} \tag{3.28}$$

Multiplying both sides of Equation 3.28 by $p_{len(Y)}$ gives Equation 3.29.

$$\frac{X}{M(len(Y))} < \frac{(\hat{E}+1)p_{len(Y)}}{2^{32}} \tag{3.29}$$

The largest value of $\hat{E}$ is 65535, and the largest value of $p_{len(Y)}$ is when $len(Y)$ is 1, which is $p_1 = 65519$. Substituting these into Equation 3.29 gives Equation 3.30.

$$\frac{X}{M(len(Y))} < \frac{65536\,65519}{2^{32}} = 1 - \frac{17}{2^{16}} < 1 - 2^{-12} \tag{3.30}$$

This shows that Equation 3.27 holds and that the new values of **approx** and **k** will be calculated correctly.

Correcting the comparison algorithm

When the comparison algorithm is considering the lengths of the two numbers a check is made to see if the difference in lengths is equal to one. If this is the case then the longer of the two numbers will be referred to as X and the shorter of the two numbers as Y.

The value in the **approx** field of X is then compared with 65535. If the approximation is greater than 65535 then it is certain that $X > Y$ and the comparison is finished, the correctness of this was shown above.

If the value in the **approx** field is less than or equal to 65535 then a copy of X is made. The length of the copy of X is decreased by one and the function **establish_length** is called. This will give new values to the **approx** and **k** fields of the copy of X. As was shown above these new values will be correct.

The copy of X will then take the place of the original X for the rest of the comparison algorithm, which will continue by comparing the values in the **approx** fields.

3.7.3 `compare`

Definition

The arguments of `compare` are two numbers of type `t_PMA`, which have had their lengths calculated using `establish_length`. It returns a positive value if the first number is greater than the second, a negative number if it is less than the second, and 0 if the two numbers are the same.

Furthermore, the magnitude of the number returned will always be equal to, or greater than, the length of the difference of the two numbers. This is used to determine the size of the result when subtracting.

Calculation

The general method has already been outlined in the introduction to this section. That is, the order of comparison is Sign, Length, Approximations, Equality, and finally, Sign of difference.

If the signs are not equal, the number returned will be plus or minus the maximum of the two number lengths plus one.

If the lengths are not equal, then first a check is made to see if the lengths differ by exactly one. If so, the procedure described in subsection 3.7.2 is carried out. Otherwise, the number returned will be plus or minus the greater of the two number lengths.

If the two approximations are equal then a globalor operation, as described in subsection 2.3.2, is performed. The value used is the difference of each residue pair, which will evaluate to false if the two residues are equal, or true otherwise. If the result of the globalor is false, then the two numbers are equal and 0 is returned.

If the two approximations differ by 2 or more, then the difference in the approximations is greater than the possible errors in the approximations. Hence plus or minus the maximum of the number length is returned. Note, this is the original, not the modified length, in the case when a number has had its length reduced.

Finally, if the approximations are within one of each other, and they are not equal, then the sign of the difference of the two numbers is calculated. A number is calculated which has residues equal to the difference of the two numbers. Then,

the function `approxCRT` is called, with the length set to the lower of the original lengths.

As in `establish_length`, the function `approxCRT` is called repeatedly with an ever decreasing length. Instead of finishing when the approximation is greater than, or equal to, $65536 - 1024$, the condition for finishing is when either the approximation is between 0 and $2^{31} - 1024$, or the approximation is between 2^{31} and $2^{32} - 1024$. These two conditions correspond with positive and negative numbers respectively.

When the sign has been found, it will be known which is the larger number. A number greater or equal to the length of the difference needs to be found. But as previously stated that is simply the last value of length used in the sign detection loop. Also note, that it is not possible to fail to find the sign, as will be shown next.

Correctness

There are two aspects to the correctness of `compare`. The first is correctly assessing which is the greatest number or that they are equal, and the second is returning a number whose magnitude is greater or equal to the length of the difference of the two numbers.

Determining which number is greater

When the signs differ, it is clear which is the larger number, as is the case when the lengths differ, once possible overestimation of length problems have been dealt with. That two numbers are equal when when all their residues are equal is also clear.

When the approximations are compared, it is assumed that a difference of two is sufficient to determine which is the larger number. This follows immediately from the correctness of `t_approxCRT` as each approximation of $X2^{32}/M$ differs by less than 1 from the real value. Hence, a difference of 2 is greater than any possible error.

When calculating the sign of the difference of two numbers a signed modular representation is used where numbers lie in the range $-\lceil M/2 \rceil \leq X_s \leq \lfloor M/2 \rfloor$. The s suffix being used to denote a signed interpretation of X.

The function approxCRT is used to determine the sign of the difference. This returns E, which is an approximation of $2^{32}X/M$. Any error in the approximation is always negative and less than 1024. When determining sign there are four ranges of E which are of significance, these are described below.

If $0 \leq E \leq (2^{31} - 1024)$ then the true value of X must lie in the range $0 \leq 2^{32}X/M < 2^{31}$. In a signed modular representation this is either a positive number or zero. It is known that the number cannot be zero, so this must represent a positive number with $0 < X_s < M/2$.

If $2^{31} \leq E \leq (2^{32} - 1024)$ then the true value of X must lie in the range $2^{31} \leq 2^{32}X/M < 2^{32}$. In a signed representation this would represent a negative number with $-M/2 \leq X_s < 0$.

If $(2^{32} - 1024) < E < 2^{32}$ then the sign of the number is not known. The true value of X must lie either in the range $(2^{32} - 1024) < 2^{32}X/M < 2^{32}$, or the range $0 \leq 2^{32}X/M < 1024$. In a signed representation this is a number with a small magnitude in the range $0 \leq 2^{32}|X_s|/M < 1024$. By reducing the number of moduli used in the approximation the magnitude of X_s/M can be increased.

If $(2^{31} - 1024) < E < 2^{31}$ then the true value of X must lie in the range $(2^{31} - 1024) < 2^{32}X/M < (2^{31} + 1024)$. In a signed representation this is a number with a large magnitude in the range $(2^{31} - 1024) < 2^{32}|X_s|/M < 2^{31}$. It will be shown that this condition can never be reached.

To show the correctness of the calculation we will first show that the initial value of $2^{32}X_s/M$ has a magnitude that is less than $2^{31} - 1024$ and so can be both represented using a signed modular representation and detected using the method being described. It will then be shown that a number which is too small to have its sign detected will still lie in the range $0 \leq 2^{32}|X_s|/M < (2^{31} - 1024)$ after the removal of a modulus.

The difference of the two numbers is formed with an initial length set as the minimum of the two number lengths. This is the length used when the approximations were compared. The difference in approximations was found to be either -1,0 or 1. The error on the approximations is less than plus or minus one as they where calculated using t_approxCRT. The starting value of $2^{32}X_s/M$ will thus have a magnitude of less than 3 which is significantly less than $2^{31} - 1024$.

If a number is too small to have its sign detected then we have seen that $0 \leq 2^{32}|X_s|/M < 1024$. When a modulus is removed, the value of M will be

divided by p_{n-1} which is less than 2^{16}. If this new modulus is named M' then the magnitude of $2^{32}|X_s|/M'$ will be less than $1024p_{n-1} < 2^{26}$. This is less than $2^{31} - 1024$ so the sign will always be detected correctly.

Length of difference

If the two numbers are of different sign, the difference between the two numbers will be greater than the magnitude of either number. However, it cannot be greater than twice the magnitude of the larger of the numbers. Increasing a numbers length by one gives scope for a p_n times increase in value. As all the moduli are significantly larger than 2, the length of the difference is certain to be less than, or equal to, the maximum of the two numbers lengths plus one.

If the lengths of the two numbers differ, or their approximations are significantly different, then the length of the larger number is returned. As the difference between the numbers is certain to be smaller than the larger number then the length of the larger number must be sufficient to store the result. However, in the rare circumstance that `establish_length` would have returned a longer length, it will be possible to have the same number stored with two different lengths. This will not cause a problem, as `compare` will automatically reduce the length of any potentially smaller or equal number, as was described above.

If a difference of sign is used to determine which number is greater then the length returned is the length at which the sign can first be detected. The size of approximation needed for the length to be detected was shown to be less than 2^{26}, which is significantly less than the approximations allowed by `establish_length` which can approach 2^{32}.

3.8 Base Extension

Base extension was discussed in section 2.4. Three methods were described which were *Individual Reconstructions*, *Mixed Radix Extension* and *Combined Individual Reconstructions*. To reduce the number of base extensions required `num_res` is always set to a power of 2. For this reason, only *Mixed Radix Extension* and *Combined Individual Reconstructions* are viable choices. While originally *Mixed Radix Extension* was used in the library, a switch to *Combined Individual Recon-*

structions was prompted by testing, which showed that it could be up to 5 times faster, despite having greater requirements for tables of data.

Base extension may occur whenever the value of a number is increased due to arithmetic operations such as addition or multiplication. It is used to ensure that the operands are of sufficient lengths to form the result. As such, the function `base_extension` is called in almost all arithmetic functions.

As the same operand may be used many times, it would not be efficient to extend it every time it is used. Imagine, for instance, incrementing a number by adding one to it. If the number is very long, it would be inefficient to keep extending the value of one every time you wished to increment it. For this reason once a number has been extended it stays that way until it is reassigned a new value, at which point the value of `num_res` is reset to be as small as possible.

3.8.1 `base_extension`

Definition

The function `base_extension` takes, as its arguments, a number of type `t_PMA` and an integer which is the new length required. It modifies the number so that the `num_res` is large enough to store a number of the length inputted. If the current `num_res` is greater or equal to the inputted length this function will do nothing. `num_res` is always a power of 2 multiplied by the number of processors.

3.8.2 Calculation

First the function determines if it a base extension is necessary, by comparing the inputted length with `num_res`. If a base extension is necessary, it repeatedly doubles `num_res` until it is greater to or equal to the inputted length.

Each processor then calculates its subset of the values y_i, using `Inverse16`. These are then broadcast by each processor in turn and the results reassembled to give a complete list of y_i on each processor.

For each new modulus p_j, each processor independently performs the following calculation.

$$X \equiv M \left(\left(\sum_{i=0}^{n-1} p_i^{-1} y_i \right) - k \right) \pmod{p_j}$$

To speed this up the values of $p_i^{-1} \bmod p_j$ are stored in `p_inverse`, k is stored in `k` and $M \bmod p_j$ is stored in the spare half of the array `Inverse16` as was described in Section 3.4.

3.9 Input and Output

The function `input` allows numbers to be inputted from the standard input (`STDIN`) and the function `output` outputs numbers to the standard output (`STDOUT`). Numbers are inputted and outputted as decimal numbers.

3.9.1 input

Definition

The function `input` takes, as its argument, a structure of type `t_PMA` which has been initialised using the function `create`. It reads a string of decimal digits from `STDIN` and assigns the value of the decimal string to the inputted structure. A minus sign immediately before the decimal digits is used to input a negative number.

Calculation

The decimal string is first buffered, so that it can be broadcast to each processor. After broadcasting, each processor calculates how many residues will be required to store the number by dividing the string's length by $\log_{10}(p_{1023})$ where p_{1023} is the last, and hence smallest, prime.

Each processor then builds up its residues by starting with $x_i = 0$ and working from the most significant digit d_{l-1}, to the least significant digit d_0. x_i becomes $|10x_i + d_j|_{p_i}$, where d_j is the digit in the current position.

Finally, when all the residues are calculated, the function `establish_length` is called to give a more accurate length and to set the `approx` and `k` fields.

Complexity

To input a string of length $O(n)$ will require $O(n)$ residues. To calculate each residue will require $O(n)$ steps each taking a constant amount of time, hence, the

total sequential time will be $O(n^2)$. Using *nproc* processors will reduce this to $O(n^2/nproc)$ but will add $O(n \log nproc)$ for broadcasting the digits.

3.9.2 output

Definition

The function **output** takes as input a number of type **t_PMA** and outputs the number to **STDOUT** as a decimal string.

Calculation

The last ten decimal digits of a number can be found by reconstructing modulo 10^{10} using the function **modCRT_32u**. These digits are then placed into an array which will later be printed.

To obtain the next ten digits, the original number is divided by 10^{10} and another modular reconstruction is performed. This process is repeated until all the residues become zero, at which point the calculation stops, and the digits are printed to **STDOUT**. To handle the problems associated with parallel I/O, only one process performs the printing. This is handled by the function **cprintf** which is described in Section B.2 of the Appendix.

To perform the division by 10^{10}, each residue has the value returned by the function **modCRT_32u** subtracted, and then is multiplied by $10^{-10} \bmod p_i$. The value of $10^{-10} \bmod p_i$ is calculated using the extended Euclidean algorithm.

Complexity

The number of reconstructions to 10^{10} will be proportional to the length of the number $O(n)$. Each of these reconstructions will take time $O(n/nproc)$ plus $O(\log nproc)$ for communication, giving a total time complexity $O(n^2/nproc) + O(n \log nproc)$.

Alternative methods

Repeatedly reconstructing a number modulo 10^{10} is not the most efficient method of performing a full CRT reconstruction. However, it does avoid the necessity to

be able to handle large integers, which a traditional method would require. It also produces a decimal output which is more familiar to the human reader.

Two alternative methods were compared using an older version of the library which ran exclusively on the Maspar MP-1. In this comparison reconstructing modulo 2^{32} was compared with performing a full CRT reconstruction.

Reconstructing modulo 2^{32} can be many times faster than reconstructing to other moduli, as all the modular reduction are performed as a by-product of overflow.

To perform the full CRT reconstruction, each processor handled 16 bits of each number. The reconstruction constants M_i were stored with 16 bits on each processor, so processor 0 had the last 16 bits of all the 1024 M_i. Also M was stored in the same way so that kM could be subtracted.

The values of y_i were calculated, and then each processor multiplied its 16 bit part of M_i by the corresponding y_i, and summed the result. From this total was subtracted, k times the local 16 bits of M.

Each processor now had a number which could be up to 42 bits long. By performing a carry operation, where each processor communicated with its neighbour, these numbers where reduced to 16 bits. By using the inbuilt `globalor` command the carry process was stopped as soon as was possible, and the number outputted.

Comparing these two methods: the modular reconstructions were faster for numbers of less than 120 moduli, but after this point the full CRT reconstruction was significantly faster with a difference of 6 times for 1024 moduli.

This method is not practical in general, though as the number of moduli used in the reconstruction is fixed. This does not affect the Maspar, as it has 1024 processors, but for other platforms this would be very inefficient for smaller numbers. By using multiple tables, this problem could be by-passed but the memory requirements at $O(n^2)$ per table would be overwhelming.

3.10 Addition and subtraction

The functions for addition and subtraction follow the same basic scheme as all the other arithmetic operations which will be described. That is, the calculation is split into several parts.

- Estimate length of result
- Base extend arguments
- Calculate residues of result
- Establish length of result

As subtraction is equivalent to adding the negation of a number, only one function is use for both addition and subtraction which is called **addsub**. The end user calls two macros called **add** and **sub**, which then tell **addsub** which operation is required.

3.10.1 addsub

Definition

The function **addsub** takes four arguments: three numbers of type **t_PMA** which are the result and two arguments, and an integer which will be 1 for subtractions and 0 for additions. It places the result of the addition or subtraction into the first argument.

Calculation

In the modular representation being used, the residues hold only the magnitude of the number, the sign being stored separately. The first part of the calculation is to determine if the magnitudes stored in the residues will need adding or subtracting.

In terms of number magnitudes, subtracting a positive number from a negative number is effectively an addition, and adding a negative number to a positive one is effectively a subtraction.

If an effective addition is being performed, the length is set to the maximum of the two lengths plus one. If an effective subtraction is being performed the function **compare** is called and the magnitude of the returned value is used as the length.

The number of residues is then determined to be the smallest power of two which is not less than the length, and the function **base_extension** is called on

both of the arguments. Note that if the arguments are already long enough the function `base_extension` will do nothing.

The memory used to store the array pointed to by `array`, is reallocated to the new number of residues, and the value of each residue is calculated as $x_i = |\pm y_i \pm z_i|_{p_i}$, where y_i and z_i are the residues of the arguments, and the signs are dependent on the exact calculation performed. For example, subtracting a larger positive number Z, from a smaller positive number Y, would require $x_i = |-y_i + z_i|_{p_i}$, as the result will be negative.

The sign of the result is then set and finally the function `establish_length` is called to assign the `length`, `approx` and `k` fields.

Correctness and Complexity

The correctness of this calculation is dependent on the length used to store the result, and the correctness of the functions `compare` and `base_extension`. These have been discussed previously, and will not be repeated.

In the best case, the time complexity will depend only on the calculation of the residues which is $O(n/nproc)$, and establishing the correctly estimated length which will require a fixed number of approximate reconstructions taking time of $O(n/nproc) + O(\log nproc)$.

However, when subtracting two numbers which have a very small difference, the time taken could be up to $O(n^2/nproc) + O(n \log nproc)$, as this is the worst case for comparison. Likewise if the arguments both need their base extended this could take time of order $O(n^2/nproc) + O(n \log nproc)$ also.

3.11 Multiplication

One of the main advantages of modular arithmetic is the ease at which multiplication can be performed. As each residue can be handled in isolation, multiplication can be performed in the same time as addition. However, it should be noted that it is significantly more likely that a base extension will need to be performed on the arguments, which will significantly increase the time complexity of the operation.

The basic structure of the multiplication algorithm is the same as the addition algorithm, starting with estimating the size of the result. This is not as straight

forward as it is for addition, and is discussed in subsection 3.11.1. Once the length has been estimated the calculation goes on as for addition, that is the arguments are base extended, the new residues calculated and finally `establish_length` is called, to give a better estimate of the length.

3.11.1 Problem of estimating new length

Predicting the length of the result of a multiplication is relatively simple when using a traditional representation, you can just add the lengths of the arguments. However, the same method does not work for the modular representation used in this library. The problem is caused by the decreasing size of the moduli used to store the number. To demonstrate this consider the set of moduli $\langle 13, 11, 7, 5, 3, 2 \rangle$. If the number 130 was to be multiplied by 131, then both numbers would be of length 2, as they are both less than $13 * 11 = 143$. Their product 17030, could not be stored using just $2 + 2 = 4$ residues as $13 * 11 * 7 * 5 = 5005$, or even 5 residues, as $5005 * 3 = 15015$. Instead 6 residues would need to be used.

In general if a number A of length $L1$ was multiplied by B of length $L2$ then the result C would have a length equal to $L1 + L2 + \delta$, where δ is an extra amount which will depend on the exact numbers multiplied. This is illustrated in Figure 3-1.

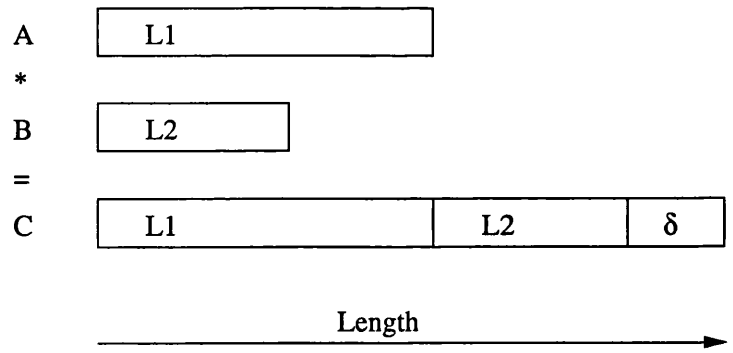

Figure 3-1: Extra length needed for multiplication

3.11.2 Calculating the extra length δ

The value of δ will depend on the exact numbers multiplied. It is the worst case that is of interest though, as an upper bound on the length is required for

`establish_length` to function correctly. The worst case would be for each of the operands to fill its length completely, so we will assume that a number of length L will equal $M(L) = \prod_{i=0}^{L-1} p_i$.

To help calculate the size of δ, a program was written which calculated all the ratios of $M(L1) * M(L2)/M(L1+L2)$, which is the ratio of the true value of multiplying $M(L1) * M(L2)$ with the supposed maximum result $M(L1+L2)$.

$$ratio(L1, L2) = \frac{M(L1)\,M(L2)}{M(L1+L2)} = \frac{\prod_{i=0}^{L2-1} p_i}{\prod_{i=0}^{L2-1} p_{L1+i}} \tag{3.31}$$

The value of $ratio(L1, L2)$ was calculated for all values such that $L1 + L2 \leq 1024$, this effectively restricted the results to those which are possible to perform using 1024 moduli.

Using these tables it was possible to calculate the value of δ for each pair of values $L1$ and $L2$ using Equation 3.32.

$$\prod_{i=L1+L2}^{L1+L2+\delta-2} p_i < ratio(L1, L2) \leq \prod_{i=L1+L2}^{L1+L2+\delta-1} p_i \tag{3.32}$$

For each value of $L1$, the largest values of $L2$ for each δ were recorded. These largest values will be referred to as $\mathrm{L2max}(L1, \delta)$. By looking at these values it was clear that for each δ, the products $L1\,\mathrm{L2max}(L1, \delta)$ are almost constant, and also that the size of these products are proportional to the value of δ. For every pair of values of $L1$ and δ the value C_{16} was calculated as defined in Equation 3.33.

$$C_{16} = \frac{L1\,\mathrm{L2max}(L1, \delta)}{\delta} \tag{3.33}$$

For each pair, the value of C_{16} was bounded by $59024 < C_{16} \leq 61256$. This enables the calculation to be worked backwards as the values of $\mathrm{L2max}(L1, \delta)$ are bounded by Equation 3.34.

$$\mathrm{L2max}(L1, \delta) > \frac{59024\,\delta}{L1} \tag{3.34}$$

If $L2$ is less than $59024\,\delta/L1$, then the value of δ must be sufficiently large as $L2 < \mathrm{L2max}(L1, \delta)$. Hence, it is possible to calculate an upper bound on δ using Equation 3.35.

$$\delta \leq \left\lceil \frac{L1\,L2}{59024} \right\rceil \tag{3.35}$$

Using Equation 3.35 the largest estimate of δ will occur when $L1 = L2 = 512$. This will give an upper bound of $\delta \leq 5$.

Using the values of L2max$(L1, \delta)$ it was possible to check the accuracy of the estimates produced by Equation 3.35. The difference between the real and the estimated value of δ was found to be 1 or 0 in all cases.

3.11.3 Theoretical Background

If P_i is the i^{th} prime then Equation 3.31 can be written as Equation 3.36 where $P_n = p_{L1-1}$.

$$ratio(L1, L2) = \prod_{i=0}^{L2-1} \frac{P_{n+L1-i}}{P_{n-i}} \tag{3.36}$$

If n is large when compared with $L2$ then each term of the product would be approximately equal so this could be changed to the following:

$$ratio(L1, L2) \approx \left(\frac{P_{n+L1}}{P_n} \right)^{L2} \tag{3.37}$$

By the Prime Number Theorem $P_n \approx n \log n$ and $P_{n+L1} - P_n \approx L1 \log(n)$ if $L1$ is small compared to n. This leads to Equation 3.39.

$$ratio(L1, L2) \approx \left(\frac{P_n + L1 \log(n)}{P_n} \right)^{L2} = \left(1 + \frac{L1 \log(n)}{P_n} \right)^{L2} \tag{3.38}$$

$$ratio(L1, L2) \approx \left(1 + \frac{L1}{n} \right)^{L2} \tag{3.39}$$

If x is small then $1 + x$ will be approximately equal to e^x.

$$ratio(L1, L2) \approx \left(e^{\frac{L1}{n}} \right)^{L2} = e^{\frac{L1L2}{n}} \tag{3.40}$$

As we are trying to approximate the number of extra moduli needed, as a power of P_n, Equation 3.40 becomes Equation 3.41.

$$ratio(L1, L2) \approx P_n^{\frac{L1L2}{n \log(P_n)}} \approx P_n^{\frac{L1L2}{P(n)+n \log\log(n)}} \tag{3.41}$$

Hence the value of δ should be approximately $\frac{L1L2}{P(n)+n \log\log(n)}$.

3.11.4 `mult`

Definition

The function `mult` takes three arguments of type `t_PMA` which are, the result and two arguments. It places the product of the two arguments into the result.

Calculation

The calculation follows the same pattern as addition. To calculate the new length the value of δ from Equation 3.35 is added to the sum of the lengths.

Correctness and Complexity

The correctness of the length comes from the calculation of δ which was described in subsection 3.11.2. As numbers are not allowed to fill their length completely, the numbers being used will be much smaller than those used to calculated δ. This is true for both the arguments and the result, however, as both arguments must be proportionally smaller than their length when they are multiplied together, there is no danger that the result will be too big for the new length.

The time complexity for multiplication is the same as addition without the possibility of a time consuming comparison. However a base extension of one or both of the arguments is very likely.

3.12 Division

The library provides separate functions for exact and general(inexact) division. The difference between the algorithms when using a modular representation is much more significant than when using a traditional representation. Using algorithms such as those proposed by Jebelean [24, 25], a speedup of 2 or 4 times is possible for exact division using a traditional representation. In a modular

representation the difference is a reduction of sequential complexity from $O(n^2)$ to $O(n)$.

3.12.1 Exact Division

To perform the exact division of $Z = X/Y$, the same general method is used as was the case for addition. That is estimate the new length, base extend, calculate the new residues, and finally call **establish_length**.

To calculate the new residues the formula $z_i = |x_i y_i^{-1}|_{p_i}$ is used. However this assumes that $y_i^{-1} \bmod p_i$ exists. If it does not then $gcd(Y, p_i) \neq 1$. As all the moduli are primes, the inverse will always exist unless Y is divisible by p_i. As X is divisible by Y then it must also be divisible p_i. In this case both X and Y are divided by p_i. To do this a modified version of **modCRT_32u** is used as is described in the next subsection.

3.12.2 Dividing by a modulus

To divide by one of the prime moduli is a special case of exact division. The values of $p_i^{-1} \bmod p_j$ are stored in the array **p_inverse**, so calculating the new residues is simple, except for the residue corresponding to the modulus being divided by. To calculate this residue, a *Reconstruction to a small modulus* is performed, however as one of the residues is missing, it has to be slightly modified to take this into account.

The reconstruction constants stored in the table **Inverse16** represent the values $M_i^{-1} \bmod p_i$ for all values of i and all numbers of moduli. However if p_j is the modulus being removed then the desired reconstruction constant is $(M_i/p_j)^{-1} \equiv M_i^{-1} p_j \bmod p_i$, which is obtained by multiplying by p_j. The value of k will also change when the number of moduli are decreased. This is recalculated again with reconstruction constants being modified as before.

Once all the residues have been calculated, **establish_length** is called using the original length.

3.12.3 `div_exact`

Definition

The function `div_exact` takes as arguments three numbers of type `t_PMA`. Into the first will be placed the result of dividing the second by the third. It is the responsibility of the user to ensure that the division is exact.

Calculation

The length of the result is set to the length of the dividend plus 2 minus the length of the divisor, unless the length of the divisor is 1 in which case the length of the result is set to the length of the dividend. A base extension of the divisor may be necessary if it is significantly shorter than the dividend.

This next part of the calculation is to ensure that an inverse can be calculated. Each residue of the divisor is checked to see if it is zero, if it is, both the divisor and the dividend are divided by the corresponding modulus, using the method described above. This may still result in a zero residue in which case the process is repeated.

The new residues are calculated using $z_i = |x_i y_i^{-1}|_{p_i}$, with each inverse being calculated using the extended Euclidean algorithm. Finally `base_extension` is called.

Correctness and Complexity

Unlike multiplication, the decreasing size of the moduli is an advantage in predicting the length of the result. Adding 2 to the difference in the length may seem to be excessive, but it should be remembered that the length of a number may be an overestimate by up to 1, and that an extra 1 is always needed even when using a traditional representation. For example dividing 99 by 3 will produce 33 a number with $2 - 1 + 1 = 2$ digits.

Although calculating an inverse is more costly than performing a multiplication or addition, as the size of the moduli is bounded, the time taken is a constant. In the best case, this results in the same time complexity for exact division as for addition or multiplication, that is $O(n/nproc) + O(\log nproc)$.

If there are zero residues this will increase the time taken to perform the exact

division. In the worst case, the divisor will be made up entirely of powers of the moduli. This would result in $O(n)$ divisions by a modulus. This would increase the complexity to $O(n^2/nproc) + O(n \log nproc)$, but it is extremely unlikely to happen.

The other possible increase in complexity would be if a base extension of the divisor was required. This would add $O(n^2/nproc) + O(n \log nproc)$.

3.12.4 General Division

A general division is significantly more complex than an exact division. This is true both for a traditional and for a modular representation. In subsection 1.1.3 several classical algorithms were presented for performing bignum calculations. Algorithm 5, was the algorithm for division, and it is this 'Knuth' style division which will be used.

Instead of building up both a quotient and a remainder, only the remainder is calculated. If the quotient is required, it can be calculated by subtracting the remainder and performing an exact division using `div_exact`.

3.12.5 The remainder operation

The purpose of the remainder operation is to calculate $X \bmod Y$. To do this X is reduced by multiples of Y until it lies in the range $0 \leq X < Y$. In each step an estimate is made of X/Y using the values stored in the **approx** fields and the lengths of the two numbers.

The values of X and Y must obey the following inequalities if the function `establish_length` and hence `t_approxCRT` has been used.

$$\frac{(approx_X - 1)}{2^{32}} M_X < X < \frac{(approx_X + 1)}{2^{32}} M_X \tag{3.42}$$

$$\frac{(approx_Y - 1)}{2^{32}} M_Y < Y < \frac{(approx_Y + 1)}{2^{32}} M_Y \tag{3.43}$$

Where $approx_X$ is the value in the **approx** field of X and M_X is the value of M constructed with length of X moduli. It follows that the value of X/Y is bounded by,

$$\frac{(approx_X - 1)M_X}{(approx_Y + 1)M_Y} < \frac{X}{Y} < \frac{(approx_X + 1)M_X}{(approx_Y - 1)M_Y} \tag{3.44}$$

and $\lfloor X/Y \rfloor$ is bounded by,

$$\left\lfloor \frac{(approx_X - 1)M_X}{(approx_Y + 1)M_Y} \right\rfloor \leq \left\lfloor \frac{X}{Y} \right\rfloor \leq \left\lfloor \frac{(approx_X + 1)M_X}{(approx_Y - 1)M_Y} \right\rfloor. \tag{3.45}$$

By using the left most term of Equation 3.45 as an approximation of $\lfloor X/Y \rfloor$ it is certain that the remainder will be positive. The accuracy of the approximation will affect how many steps the division takes. The error in approximating $\lfloor X/Y \rfloor$ is at most the difference in the left and right terms of Equation 3.45.

$$Error \leq \left\lfloor \frac{(approx_X + 1)M_X}{(approx_Y - 1)M_Y} \right\rfloor - \left\lfloor \frac{(approx_X - 1)M_X}{(approx_Y + 1)M_Y} \right\rfloor \tag{3.46}$$

Removing the truncations and simplifying, this becomes Equation 3.47.

$$Error - 1 \leq \frac{2(approx_X + approx_Y)M_X}{(approx_Y + 1)(approx_Y - 1)M_Y} \tag{3.47}$$

As $\frac{X}{Y} > \frac{(approx_X - 1)M_X}{(approx_Y + 1)M_Y}$, this can be expressed in terms of $\frac{X}{Y}$.

$$Error - 1 \leq \left(\frac{2(approx_X + approx_Y)}{(approx_X - 1)(approx_Y - 1)} \right) \frac{(approx_X - 1)M_X}{(approx_Y + 1)M_Y} \tag{3.48}$$

$$Error < \left(\frac{2(approx_X + approx_Y)}{(approx_X - 1)(approx_Y - 1)} \right) \frac{X}{Y} + 1 \tag{3.49}$$

From Equation 3.49 it can be seen that when $X >> Y$ the accuracy of the estimation of $\lfloor X/Y \rfloor$ will depend on the size of the approximations of X and Y. The accuracy of the approximation defined as the true value divided by the error is approximated in Equation 3.50.

$$Accuracy \approx \frac{approx_X\, approx_Y}{2(approx_X + approx_Y)} > \frac{MIN(approx_X, approx_Y)}{4} \tag{3.50}$$

As the smallest value of an approximation is around 2^{16} and the maximum size of an approximation is 2^{32}, the accuracy should be between 2^{14} and 2^{30}. Which

represents a decrease in the size of the X of between 14 and 30 bits.

When the value of X approaches the value of Y the accuracy decreases. When the estimated value of $\lfloor X/Y \rfloor$ becomes zero, the main loop stops and a check is made to see if $X < Y$, if not Y is subtracted. The error at this point is assumed to be at most 1. To show that it is not possible to leave the main loop with an error of 2 or more, the smallest value of X/Y which can cause an error of 2 will be calculated, starting by setting *Error* to 2 in Equation 3.49.

$$2 < \left(\frac{2(approx_X + approx_Y)}{(approx_X - 1)(approx_Y - 1)} \right) \frac{X}{Y} + 1 \tag{3.51}$$

$$\frac{X}{Y} > \frac{(approx_X - 1)(approx_Y - 1)}{2(approx_X + approx_Y)} \tag{3.52}$$

As this bound is the same as was used in the calculation of *Accuracy*, the smallest value of X/Y at which an error of 2 can occur is approximately 2^{14}. If the value of X/Y is less than this amount the error will be at most 1.

Normalisation

While the value of X will change in each loop the value of Y is fixed. Hence the value of $approx_Y$ is fixed throughout the calculation. In the 'Knuth' style division algorithm (Algorithm 5) a normalisation step is used to ensure that the most significant digit of the divisor is greater than half the number base. Similarly it is an advantage for the approximation of Y to be as large as possible as a small value would decrease the accuracy of the approximation in each end every step.

To normalise the value of Y the value of $approx_Y$ first has one added, to it to ensure that it is an overestimate, and then it is repeatedly doubled until it is greater than 2^{31}. The values of both X and Y are then scaled by the appropriate power of 2.

3.12.6 `r_div`

Definition

The function `r_div` takes as its arguments three numbers Z, X and Y of type `t_PMA`. It will set $Z = X \bmod Y$. If Y is negative then Z will lie in the range $Y < Z \leq 0$.

Calculation

Unlike the other arithmetic operations the length of the result is not calculated at the beginning of the calculation. Instead the values of X and Y are duplicated so they may be modified within the function, with the value of X being returned as Z.

The first stage of the calculation is to normalise the numbers as described above. It should be noted, however, that the length of Y could be increased by this operation, which is not desired. To by-pass this problem of overestimating length, the function `t_approx` is called with the length set to the original length of Y. This ensures that the approximation of Y is the size required.

When approximating the value of $\lfloor X/Y \rfloor$, the value of M_X/M_Y is required. This is referred to as M^* and is initially calculated as the following product.

$$M^* = \prod_{i=length(Y)}^{length(X)-1} p_i \tag{3.53}$$

The main loop has three main sections. In the first, the value of $\lfloor X/Y \rfloor$ is approximated. In the second, the value of X is reduced, and in the third, a new value of M^* is calculated.

To approximate $\lfloor X/Y \rfloor$ the value of M^* is first multiplied by $approx_X - 1$. This forms the top of the right hand side of Equation 3.54.

$$\left\lfloor \frac{(approx_X - 1)M_X}{(approx_Y + 1)M_Y} \right\rfloor = \left\lfloor \frac{(approx_X - 1)M^*}{approx_Y + 1} \right\rfloor \tag{3.54}$$

The value of $approx_Y + 1$ is already known. So to perform the exact division a reconstruction to a small modulus is used to give the value of $(approx_X - 1)M^* \bmod (approx_Y + 1)$. This value is then subtracted and an exact division is performed.

The approximation of $\lfloor X/Y \rfloor$ is then multiplied by Y, and the product is subtracted from X, using the functions described previously.

The last part of the loop corrects M^* which will have altered, if the length of X has changed. For each modulus which is no longer required to reconstruct X, the value of M^* must be reduced, using the division by a modulus method, described in subsection 3.12.2.

When an approximation of $\lfloor X/Y \rfloor$ is equal to zero, the loop is exited and a

comparison is made between X and Y, if $X \geq Y$ then Y is subtracted from it.

Finally, the remainder is un-normalised by an exact division by the power of 2, that was used during normalisation.

Computational Complexity

If the length of X is n, and the length of Y is m, then we hope to calculate the complexity of division in terms of these two variables. As there are many stages to division these have been split up to simplify the discussion.

M^*

The initial calculation of M^* involves multiplying together $n - m$ moduli. This will produce a number of length approximately $n - m$. To avoid later base extensions, the value of M^* is calculated with a number of residues sufficient to store X.

The multiplication is done using the function `scale`, which multiplies a number of type `t_PMA`, by an integer of bounded length. Importantly, it does not create a new number, but it modifies the existing residues. Hence, the number of residues is preserved.

To scale each of the n residues will take time equal to $O(n/nproc)$ and to establish the length will take $O((n-m)/nproc) + O(\log nproc)$, giving each step a complexity of $O(n/nproc) + O(\log nproc)$. The total complexity for all $n - m$ steps becoming $O((n-m)n/nproc) + O((n-m)\log nproc)$

During the main loop, M^* will be reduced. This will happen a maximum of $n - m$ times, with each step having the same complexity as scaling, hence the total complexity for all calculations involving is $O((n-m)n/nproc) + O((n-m)\log nproc)$.

Main loop

The number of iterations needed to perform a division, will be proportional to the difference in the lengths of the two numbers taking part in the division, that is $O(n - m)$.

To form the estimated value of $\lfloor X/Y \rfloor$ several steps are required. First the current value of M^* is duplicated so that it can be modified without effecting

the original. This number is then scaled by the **approx** field of X and a reconstruction to a small modulus is performed. The result of this reconstruction is then subtracted from each modulus and an exact division is performed using **un_scale** which changes the existing value and hence preserves the number of residues. Each of these steps involve a fixed number of operations on each modulus, plus a fixed number of approximate reconstructions. This gives an overall complexity per step of $O(n/nproc) + O(\log nproc)$.

Once the approximation of $\lfloor X/Y \rfloor$ has been calculated, it is multiplied by Y. In the first step the value of Y may have to be base extended for the result to be calculated. However, once this has happened no subsequent base extensions will be required, as the value of X which is being estimated will decrease. If a base extension is necessary, it will cost time of order $O((n-m)m/nproc) + O(m \log nproc)$ (see subsection 2.4.4). It will not be necessary to base extend the estimate of $\lfloor X/Y \rfloor$ as this is calculated using n moduli throughout. Without a base extension, each multiplication will take take the same time as scaling, and hence does not add to the complexity.

The last stage is the subtraction of the estimate of X from the true value. The cost of subtraction is partially affected by the number of comparisons needed to establish the sign. If the numbers are close enough to need many comparisons however, then the estimate of X must be very accurate. This will lead to less iterations of the main loop and hence the cost of establishing the sign is more than canceled out.

In all the $O(n-m)$ loops will have a maximum complexity of $O(n(n-m)/nproc) + O((n-m)\log nproc)$ plus a possible base extension of $O((n-m)m/nproc) + O(m \log nproc)$.

Overall complexity

The complexity of all calculations involving M^* is the same as the complexity of the main loop if a base extension is not required. In this case the overall complexity is shown in Equation 3.55.

$$O(n(n-m)/nproc) + O((n-m)\log nproc) \tag{3.55}$$

If a base extension is required, it will have the complexity shown in Equa-

tion 3.56.

$$O(m(n-m)/nproc) + O(m \log nproc) \tag{3.56}$$

Combining Equation 3.55 and Equation 3.56 will give the overall complexity when a base extension is required.

$$O((n-m)(n+m)/nproc) + O(n \log nproc) \tag{3.57}$$

3.12.7 Other division functions

As well as `r_div`, the functions `q_div` and `qr_div` return either the quotient, or both the quotient and the remainder. The quotient is calculated using a subtraction, and an exact division, which should be significantly less costly than calculating the remainder in the first place.

3.13 Conversion to 32 bit primes

3.13.1 Why are 32 bit primes needed?

The main reason for using 32 bit primes is the increase in the range of numbers that can be represented. Their are 6542 primes less than 2^{16} the product of which has 28,305 decimal digits.

There are many more primes less than 2^{32}. One source [4] lists their number at 203 million. They are also closer together relative to their magnitude. Hence using 32 bit primes gives scope for significantly bigger numbers and increased efficiency.

3.13.2 Choice of moduli

The same system of choosing the largest prime as p_0, the next largest as p_1 and so on, is used for the 32 bit primes as was the case with the 16 bit primes. To allow errors to be bounded, an absolute limit of 65536 primes has been set. This may seem rather small compared to 203 million, but as the size of the tables grows with the square of the number of moduli, even this number is beyond the storage capacity of any system that the library has been tested on.

3.13.3 Datatype

The datatype with 32 bit moduli is shown in Table 3.3. The differences between this datatype and the one shown in Table 3.1, are that the `array` field now points to an array of 64 bit integers, and the approximation stored in the `approx` field is now also 64 bit. The 64 bit array allows a modular multiplication to be performed using a 32 bit modulus. The approximation is now stored using 64 bits as some functions need accuracy proportional to the size of the moduli.

Field	Type	MPL Type
neg	`int`	singular
array	`unsigned long long*`	plural
num_res	`int`	singular
length	`int`	singular
approx	`unsigned long long`	singular
k	`int`	singular

Table 3.3: Fields of structure T_PMA using 32 bit moduli

3.13.4 Tables of pre-calculated data

The data stored for the 32 bit moduli is much the same as for the 16 bit moduli except that the size of all the data has doubled due to the use of 32 and 64 bit integers where 16 and 32 bit integers were used previously. The only major change is the loss of the `p_inverse` table.

During base extension only half of the values in `p_inverse` are used. Conveniently, only half the values in `Inverses32` exist, so the two tables are merged together. Unfortunately in the 16 bit tables the values of $M \bmod p_i$ were stored. By not storing these values additional calculations are required during base extension, but it was decided that halving the size of the tables was worth the extra calculations.

The size of the tables are shown in Table 3.4. As the total size of the tables grow quadratically with the number of moduli, increasing the number of moduli to 65536 has a dramatic effect. The total plural data becomes over 16 gigabytes which is more memory than is available on any of the systems the library has been tested on. Therefore, each platform has its own maximum number of moduli

which is dependent on available memory.

Field	Size (Bytes)	MPL Type
`Inverses32`	$4n^2$	plural
`Primes32`	$4n$	plural
`two_64`	$4n$	plural
`p_inv`	$8n$	plural
`All_Primes32`	$4n$	singular
`M_64`	$8n$	singular
`total plural`	$4n^2 + 16n$	plural
`total singular`	$12n$	singular

Table 3.4: Size of tables using n 32-bit moduli

3.13.5 Coping with different size datasets

Even if it is possible to store several gigabytes of tables in memory this would be a waste of resources if only a small proportion of the data is used. For example, if the maximum length of any number being used is only 1000 moduli, there would be no point loading tables for 65536 moduli, which would be 4096 times bigger.

Instead of loading tables for the maximum number of moduli by default, the tables for 1024 moduli are loaded during initialisation.The function `load_inv` can be used to set the number of moduli and the function `clear_inv` can be used to remove the current set of tables.

As the user may not know the size of numbers required in advance, the loading of tables is done on demand. This is handled by the function `inverses32`, which is called in place of the array `Inverses32`. If a request is made for an inverse which is out of the range of the currently loaded tables, a table of sufficient size is loaded. The tables are stored in separate files, with the number of moduli starting at 1024 and doubling up to the maximum that can be stored in memory.

3.13.6 Changes to 16 bit code

All of the functions work in exactly the same way using 32 bit moduli as they did using 16 bit ones. The main differences in the functions are that the datatypes used are all twice as large as before. When the previous datatype was 16 or

32 bits, this did not cause a problem, as all the systems used supported 64 bit integers. Problems were only caused when previously 64 bit integers had been used, as none of the systems used supported 128 bit integers. In most cases the size of the sub-results could be kept down by reducing them in each step of the calculation, but in one case a custom 128 bit multiplication and subsequent modular reduction function had to be written.

What did not cause a problem was maintaining sufficient accuracy for the functions to perform correctly. The reason for this is simple. As the accuracy of each calculation is increased from 32 to 64 bits, but the number of residues has only increased from 2^{10} to 2^{16}, 26 bits of accuracy are gained in a standard approximate reconstruction.

During multiplication, the problem of predicting the new length is also removed. The number of extra moduli required was shown to be approximately $\frac{L1L2}{P(n)+n\log\log(n)}$, where $L1$ and $L2$ are the lengths of the numbers being multiplied and $P(n)$ is the size of one of the moduli. As there are approximately 203 million primes less than 2^{32}, and as $L1 + L2 < 65536$, the worst case is $\frac{32768*32768}{2^{32}+203000000(2.95)} \approx 0.219$, which is significantly less than 1. This was double checked by testing all possible values and the result was found to hold. Hence during multiplication δ is always taken to be 1.

In general, restricting the values of $L1$ and $L2$ to be less than $\sqrt{P(n)}$ will ensure that δ is less than 1. As length is defined in terms of $P(n)$, the numbers to be multiplied must be less than a value which is approximately $P(n)^{\sqrt{P(n)}}$.

3.14 Conclusions

In this chapter two libraries have been discussed which allow arithmetic to be performed using integers stored using a modular representation. What make these libraries unique, is that as well as storing an array of residues, a length and an approximation is stored. This allows tasks such as comparison to be performed, without the need to fully reconstruct the numbers.

By using a base extension, the number of residues used to store each number can be kept to a minimum, with extra residues being generated as required. This coupled with an approximate reconstruction form the core of the library.

It has been shown that the libraries can perform all the basic arithmetic tasks

including comparison, inexact and general division. It has also been shown that significant complexity advantages can be found when performing multiplications and exact divisions.

Chapter 4

Arithmetic using a fixed number of residues

4.1 Introduction

In Chapter 3, a library of functions was presented which allowed general arithmetic functions to be performed on integers, which happened to be stored in a modular representation. In this chapter an extension to the previous library is described in which the properties of modular arithmetic can be exploited directly.

A new datatype is used in which each number is stored as an array of residues together with the number of residues stored. All calculations are made assuming that the result of a calculation will fit into the number of residues chosen. It is the responsibility of the user to ensure that this is the case. The range of functions is also restricted. Specifically, there is no comparison and no general division.

The trade off for these restrictions is that the calculations are simplified significantly, and more importantly from a parallel computing point of view most functions dealing only with this new datatype can be performed without any communication.

As this is an extension of the previous library, there is a 16 bit modulus and a 32 bit modulus version of all the functions.

Field	Type	MPL Type
array	`unsigned int*`	plural
num_res	`int`	singular

Table 4.1: Fields of structure T_LPMA

4.2 Datatype

Numbers are stored using the datatype `t_LPMA`. The extra L stands for locked, as the number of residues is locked when the number is created. No base extension function is provided for numbers of this type, though it is possible to convert the number to type `t_PMA`, base extend it, and then convert it back.

As is shown in Table 4.1 this datatype is a stripped down version of `t_PMA`. Only the residues themselves and their number is stored. With 32 bit moduli the array is of type `unsigned long long`, the 16 bit version being shown.

When using this datatype, no record is kept of the length of the number, hence there is no `length` field. Without this, `approx` has no meaning either. Also, as intermediate results may be larger than the product of moduli, `k` would be of no use. The exclusion of `neg` is due to the difficulty of subtracting without comparison, using this datatype it is assumed that numbers range between $-\lfloor M/2 \rfloor$ and $\lfloor M/2 \rfloor$, not 0 and $M-1$. This allows subtraction to be performed without the need for comparison, but halves the effective range.

4.3 Tables of pre-calculated data

As no approximate reconstructions or base extensions are performed when using this datatype, the only part of the tables needed is the array `Primes16/32`. The full tables are used however, when the type `t_PMA` is used, which includes the input and output functions, where type `t_PMA` is used as an intermediate form.

4.4 Calculating bounds

It is envisaged that the general method used with this datatype is as follows.

- Calculate Bound using `t_PMA`

- Convert starting values to t_LPMA
- Perform calculation using t_LPMA
- Convert back to t_PMA

To this end the function L_get_bound will take a number of type t_PMA and return a number of residues sufficient to store the number using type t_LPMA. As the only difference between the two representations is that negative numbers are encoded in the residues of t_LPMA, the function simply returns the num_res field, unless the length of the number equals the number of residues. If the length does equal the number of residues, then it is compared with either $2^{31} - 1024$ or $2^{63} - 65536$. If it is found to be greater or equal, then the number of residues are doubled. The reason for these particular bounds is that they allow the sign to be detected when a number of type t_LPMA is converted back to t_PMA as will be shown in section 4.5.

Once the bound is known it is possible to create the numbers of type t_LPMA using the function L_create, which takes the number of residues as its argument, and returns the new number. To destroy a number, the function L_destroy is called. To give the number a value it is possible to either assign it the value of another number of type t_LPMA using L_assign, assign it the value of a 32 bit integer using L_assign_32u, or assign it the value of a number of type t_PMA as described below.

4.5 Type conversion

4.5.1 t_PMA → t_LPMA

Converting from t_PMA to t_LPMA can be as simple as duplicating the array and the number of residues. However, it is possible that more residues are required for the calculation in t_LPMA, so a base extension may be necessary. As there is no sign bit in the new representation, negative numbers will have to have all their residues negated.

4.5.2 L_convert_to

The function L_convert_to takes as arguments a structure of type t_LPMA and a number of type t_PMA. It will convert the number of type t_PMA to t_LPMA, with a number of residues which was determined when the structure of type t_LPMA was created.

4.5.3 t_LPMA → t_PMA

There are two main problems when converting a number of type t_LPMA to type t_PMA. These are determining the length and determining the sign of the number. If, during a comparison, it is not possible to separate two numbers using the **length** or **approx** fields, then the two numbers are subtracted and the sign of the difference is measured. As a by-product of this calculation an upper bound is placed on the length of the difference.

There is a difference between the numbers expected during a conversion and the numbers expected during a comparison. In a general comparison, it is unlikely that the length of the difference between two numbers is significantly less than the lengths of the individual numbers. This makes it practical to search for the length of a number starting from the original lengths of the numbers.

However it is extremely unlikely that a calculation will result in a number which is close to the bound calculated. The length of the result may be less than half the length of the bound. In this case, testing every possible length between the upper limit of the number of residues in the number and the true length could involve $O(n)$ steps.

An alternative method of determining number length was presented in subsection 2.3.4. It is called *A Probabilistic Binary Search.* This method involves a binary search followed by a probabilistic confirmation function. The number of steps needed to find the probable length is $O(\log n)$.

4.5.4 A Probabilistic Binary Search

Binary search

The first stage of this algorithm is a binary search based on an approximate reconstruction. If the approximation is greater or equal to $2^{32}-1024$ or $2^{64}-65536$

then it is assumed that the length of the number is less than the number of moduli used in the reconstruction. If the approximation is smaller than this number then it is assumed that the length of the number is greater or equal to the number of residues in the reconstruction.

The binary search will terminate with what is assumed to be the length of the number. Note that in fact the length returned will be slightly greater than the true length but this can be corrected later using establish length.

The chance that the length returned will be too small was given in Equation 2.41, which is repeated below, where n is the number of residues and d is the precision of the estimation.

$$1 - \left(1 - n2^{-d}\right)^{\lceil \log_2(n) \rceil}$$

When 1024 16-bit moduli are used the maximum chance of an error occurring is $2.384 * 10^{-6}$, and when 65536 32-bit moduli are used the maximum chance of an error occurring is $5.68 * 10^{-14}$.

Confirmation

In the next stage, an attempt is made to see if the predicted length is sufficient to reconstruct the number. The check that is made is to reconstruct the number to a small modulus using first all the moduli, and second the predicted length moduli. If the two reconstructions differ then the length is not sufficient, but if they do not differ then it is probable that the length is sufficient.

The moduli used are selected using the `random` function. This returns a pseudo-random number between 0 and $2^{31} - 1$. The test is repeated a fixed number of times, which is defined in the library header files and is currently set at 10. The chance that a number will pass the test is 1 in the least common multiple of the tested moduli. If all the numbers were relatively prime the least common multiple would be approximately $(2^{30})^{10} = 2^{300}$, where 10 is the number of moduli tested. To test the effect of using numbers which may not be relatively prime, an average of a thousand least common multiples was taken. This average had 290 bits giving a probability of the confirmation function failing of 2^{-289}.

Repeated approximations

If it is found that a mistake had been made in the binary search, then the length is found using the standard repeated approximations method. While time consuming, this is certain to find the true length. It would have been possible to perform the binary search using modular reconstructions as proposed in subsection 2.3.4. However, the chance of the binary search failing is very small, so the time taken is not of great concern.

Sign Detection

Once a length has been found, the sign of the number can be tested. It is known that the approximation returned by `approxCRT` must be less than either $2^{32}-1024$ or $2^{64}-65536$, depending on the size of the moduli. This ensures that the sign can be determined, as errors in approximations are always negative and less than either 1024 or 65536.

As half the potential values are negative, and half are positive, a number will be deemed negative if its approximation is greater than 2^{31} or 2^{63}. If the number of residues used is larger than the true length of the number, then the calculated length will be the largest at which the sign can be determined. This was discussed in Section 3.7 where it was shown that the accuracy of the approximations was sufficient for the sign to be determined correctly. However, if the number of residues is equal to the true length of the number, then the range of possible values must be considered.

If the function `L_get_bound` has been used then the largest approximation allowed when `length` = `num_res` is either $2^{31}-1024-1$ or $2^{63}-65536-1$. As the `approx` field has a maximum error of 1, the largest true values are $2^{31}-1024$ or $2^{63}-65536$. As the error is always negative, the largest approximation of a positive value will be $2^{31}-1024$ or $2^{63}-65536$. However, when the number is negative the error which is less than 1024 or 65536, will produce a least negative approximation which is greater than 2^{31} or 2^{63}.

Correcting the length

If the number is negative, all the residues will be negated as `t_PMA` uses a sign bit. Once this has been done `establish_length` is called to give a more precise

length measurement.

4.5.5 L_convert_from

The function L_convert_from takes, as its arguments, a structure of type t_PMA which has been initialised using create, and a number of type t_LPMA. It will convert the number of type t_LPMA to t_PMA, with the same number of residues as the original number.

4.6 I/O

When using numbers of type t_LPMA the functions input and output are replaced with L_input and L_output. Both of these functions use their t_PMA equivalent functions to perform the conversion into and out of decimal form. While some speedup could be obtained by using custom functions, it is not envisaged that these functions will be heavily used.

4.7 Available functions

Only a subset of the arithmetic functions available when using t_PMA can be used with numbers of type t_LPMA. These are listed below:

- L_is_zero
- L_is_equal
- L_negate
- L_add
- L_sub
- L_mult
- L_div_exact
- L_scale

- `L_unscale`

A full list of the function prototypes is provided in Appendix B.3. Importantly, by removing the need to keep approximations and to continuously monitor length, the complexities of the above functions are reduced. For all apart from the functions `L_is_zero` and `L_is_equal`, there is no communication required. This reduces the complexities of these functions to $O(n/nproc)$ where n is the number of residues. For `L_is_zero` and `L_is_equal` communication is required adding $O(\log nproc)$ time.

4.8 Restrictions in use

The cost of such speed and restriction in communication, is a certain lack of flexibility. As can be seen from the list above, there is no comparison or general division. Also missing is reconstruction to a small modulus, though it would be perfectly possible to implement this. The reason it has not been implemented is that many of the calculations for which this part of the library was intended, may involve intermediate results above the calculated bound. In these cases, a reconstruction to a small modulus may give incorrect results.

Another restriction, is that exact division cannot be performed when the divisor is a multiple of one or more of the moduli. Again, this could be fixed by using a custom reconstruction to a small modulus, but, also again, this may give incorrect results if an intermediate result exceeded the bound.

Finally it should be noted that it would make no sense to try and perform arithmetic using numbers stored with differing numbers of moduli. Any attempt to do this will result in the program terminating with an error.

4.9 Conclusions

In this chapter an extension to the previous library of functions of Chapter 3 has been described. This extension uses a new datatype `t_LPMA` in which the number of residues is fixed(locked) at the time of creation.

By removing the `length` and `approx` fields, it is possible to perform the majority of calculations without the need for communication between processors.

What is more, the complexity of the arithmetic operations is fixed at $O(n/nproc)$. However there is no general division or comparison, so only calculations in which the residues can be calculated independently can be performed.

An implementation of the probabilistic length algorithm of subsection 2.3.4 is described which is used for conversion from type **t_LPMA** to type **t_PMA**.

Chapter 5

Testing and Benchmarking

5.1 Introduction

In this chapter is a description of the testing and benchmarking performed on the library.

In Section 5.2, the range of hardware used to test the libraries is described. This includes a cluster of workstations, two shared memory machines and a SIMD machine. The cluster of workstations is capable of communicating using both Ethernet, and also a custom electronic board. The portable libraries MPI and AFAPI are used allowing the code to be run on the shared memory machines.

Three basic tests are described, in Section 5.3, these test the three fundamental algorithms used in the library, which are the *Approximate CRT Reconstruction*, *Combined Individual Reconstructions*, and *Reconstruction to a Small Modulus*. These algorithms are tested using the library functions **establish_length**, **base_extension**, and mod_CRT. Using the results of these tests, the performance of the libraries when performing larger arithmetic tasks can be predicted.

In Section 5.4, the basic arithmetic operations are both tested and timed. These tests include multiplication, division and addition/subtraction. The tests are intended to show the correctness of the calculations made using the library, as well as providing timing data.

5.2 Platforms

5.2.1 A Linux cluster

The main platform used for developing the library, was a cluster of four PCs running Linux. Each of these machines has 32 megabytes of RAM and they are connected using 10 Mbit/s UTP Ethernet. To communicate using Ethernet, the LAM MPI library is used. However, this is not ideally suited to the libraries as Ethernet is a high latency communication mechanism. The libraries rely on many small messages being passed with the most common operation being a parallel reduction. Each reduction involves a single integer per machine, and as these are passed individually, messages will be of size 32 or 64 bits. What is needed is a communication mechanism with a low latency. It was for this reason, that an electronic circuit board was constructed.

TTL_PAPERS

The TTL_PAPERS circuit board is a Transistor Transistor Logic (TTL) circuit board, which is an implementation of the Purdue Adapter for Parallel Execution and Rapid Synchronisation (PAPERS). The circuit board is described in detail in [11, 9]. The board connects together four computers, via their parallel ports.

The circuit design is shown in Figure A-1, which is in Appendix A. It consists of two main sections, one of which performs synchronisation, and the other, aggregate communication. The other parts of the circuit are the power regulator and the status lights.

The circuit was built using strip board and can be seen in Figure A-2 which is also in Appendix A. It differs slightly from the original design due to the availability of some of the components, but retains the same functionality and most importantly can use the same libraries, which are called AFAPI.

The Aggregate Function API (AFAPI) [10], is a library of communication primitives originally developed for the PAPERS design. The libraries provide the same types of functions which are available in MPI, but while MPI relies on message passing, AFAPI relies on broadcasting and synchronisation. However the AFAPI libraries are not just restricted to the PAPERS circuit boards, but also can be used on a shared memory computer.

Comparison of Ethernet/MPI and TTL_PAPERS/AFAPI

To test the relative speeds of the two methods of connecting the Linux boxes, the time taken to broadcast messages of different sizes was measured directly by transferring a 1 megabyte file between the computers. By breaking the communication into different size transfers, the effect of latency can be measured. The results are shown in Table 5.1, where the time taken is for a single broadcast.

Size(Bytes)	TTL_PAPERS	MPI
1	15 μs	4.0 ms
4	65 μs	4.1 ms
16	270 μs	4.0 ms
64	1.0 ms	4.0 ms
256	4.2 ms	4.1 ms
1K	17 ms	4.3 ms
4K	71 ms	15 ms
16K	280 ms	61 ms
64K	1.9 s	200 ms
256K	4.3 s	750 ms
1M	17 s	2.7 s

Table 5.1: Time for a single broadcast under normal load

As can be seen in the table the time taken to perform a broadcast using the TTL_PAPERS circuit board is directly proportional to the size of the message. This is because each block of 4 bits is transfered as a separate action, with a synchronisation between each. The minimum time for a communication would thus be around 8 μs, although this was not tested.

The minimum time taken to perform a broadcast over Ethernet using the MPI libraries was 4 ms. It took the same 4 ms for messages of between 1 byte and 1 kilobyte. For messages of 4 kilobytes and above the time taken increased in proportion with the message size.

The Ethernet results are consistent with a TCP Maximum Segment Size of 1460 bytes, with the time taken to send each datagram varying only slightly with the message size.

For large messages the Ethernet is significantly faster than the TTL_PAPERS circuit board. The Ethernet was 6 times faster at broadcasting a 1 megabyte message.

For small messages the TTL_PAPERS circuit board was significantly faster than the Ethernet. In the libraries the typical message size is 32 or 64 bits. For a 4 byte message the TTL_PAPERS circuit board is 60 times faster than the Ethernet.

5.2.2 Shared Memory Machines

As the libraries used by the Linux machines use portable communication libraries it is possible to use the same source code for any platform with a corresponding library. Two shared memory machines were tested, both of which had AFAPI, and MPI libraries.

One machine was a 4 processor Solaris server and the other was a 20 processor SGI machine. Both of the computers were tested using only 4 processors, to allow direct comparison with the Linux machines. As these machines are both used by a large number of people, the results are highly sensitive to the current load on the machine.

5.2.3 Maspar

The library was originally designed to be run on a Maspar MP-1. The Maspar MP-1 is a massively parallel SIMD computer, the machine used having 1024 processors. This gives one modulus per processor when using the 16 bit library. However, this machine developed a hardware fault, so a new machine had to be found. The new machine is a Maspar MP-2 and has 4096 processors.

As the new machine is capable of performing 4096 calculations at the same time, it makes sense to use at least 4096 moduli, for both the 16 and 32 bit libraries. However, it is not possible to use more moduli than this, as the tables required are too large to fit into each processor's 64 kilobytes of memory.

Communication Hardware

The major advantage of the Maspar is the speed of communication. Broadcasting and globalor operations are both machine primitives, and take less time to perform than many arithmetic operations. Parallel reduction is performed using the built in *xnet* which is a grid of neighbour to neighbour links. The *xnet* connects the processors in a two dimensional grid. This is very fast, giving reduction times,

which are close to optimal. Timings of the basic arithmetic and communication operations can be found in [29].

Modifications of the library

The library is slightly modified for the Maspar MP-2. As the number of moduli is equal to the number of processors, there is no need for base extension. This can save a lot of time when the numbers become large. If a method could be found to reduce the size of the tables, more moduli could be used, and base extension would once again be needed.

The Array Control Unit (ACU) controls the 4096 array processors. This is a more conventional processor than those used in the array, and has significantly more memory. However, most of this memory is taken up with the program, leaving only a few hundred kilobytes to store data. To avoid using this valuable space, all of the singular tables as described in subsection 3.4.4 are distributed between the processors, with a `proc` statement being used to broadcast the data.

5.3 Costs of Basic operations

In order to predict the performance of the libraries, the time taken to perform the basic operations must be known. These basic operations are establishing the length of a number, extending the number of residues used to store a number, and reconstructing the number to a small modulus.

5.3.1 Methods used to gather data

The results of all the tests are gathered in the same way, first a small power of two is assigned to a variable. This power of 2 is then, either used directly as the argument to the function being tested, or it is used to ensure that the numbers tested are of sufficient size.

Once the arguments have been calculated, the test is performed a number of times. The total, user and system times taken to perform the test are measured using the timing functions provided by the library. These functions in turn call `gettimeofday` and `getrusage` which return the time in microseconds.

The largest of these three times is taken, as the time taken to perform the test. This should be the total time, but this is not always the case. It is not possible to use the user time alone, as communication costs may not be included in this time.

If the time taken is less than half a second, then it is rejected as being too short to give an accurate result. If this happens the number of times the function is called is doubled, and the test is repeated. If the time taken is larger than one second, then the result is accepted, but the number of times the function will be called in the next test is reduced.

After each successful test the power of two is squared, any other arguments are recalculated, and another timing loop is started. This continues until there is not enough space to store the power of two.

5.3.2 Single processor results

A single processor version of the library has also been written. The results obtained using this library are shown next to those obtained using the parallel libraries.

While the removal of communication simplifies the single processor code, the same algorithms are used throughout. This allows fair comparison with the results obtained using the parallel libraries.

5.3.3 Establish length

The function `establish_length` is used in almost all of the arithmetic functions described in Chapter 3. To establish the length of a number, at least two *Approximate CRT Reconstructions* are needed. One is used to confirm that the length of the number is correct, and another is needed to increase the accuracy of the approximation to the standard of `t_approxCRT`.

In this test the function `establish_length` was called on the power of two, which is calculated as described above. As the length will have already been correctly set when the power of two was squared, only two *Approximate CRT Reconstructions* are required.

The fact that the moduli are all close to powers of 2 has little effect on the timing, as the first stage of the *Approximate CRT Reconstruction* is to calculate

$y_i \equiv x_i Inv_i \bmod p_i$. Past this point in the calculation the size of the remainder x_i will have no effect.

Even this small effect on the timing will be lost once the power of two is 64 or greater. The largest modulus used is 65521, using this modulus the remainder of 2^{16} is 15, of 2^{32} is 225 and of 2^{64} is 50,625.

Linux Cluster

The results of the test when using the Linux cluster is shown in Table 5.2. The columns marked SINGLE represent the results obtained using a single workstation. The columns marked AFAPI and MPI represent the results obtained using four workstations.

Size	16bit			32bit		
(Bits)	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
16	11	428	4,142	14	309	4,297
32	17	556	4,316	23	747	4,522
64	25	656	3,885	32	1,172	4,663
128	43	656	4,113	53	1,324	5,331
256	76	657	5,482	87	1,512	4,014
512	140	912	3,917	155	1,460	4,042
1,024	270	857	3,954	296	1,712	4,883
2,048	545	952	4,831	595	1,706	4,196
4,096	1,159	1,074	4,227	1,240	2,033	4,395
8,192	2,371	1,291	4,576	2,474	2,010	5,571
16,384	-	-	-	4,850	2,880	7,068
32,768	-	-	-	9,894	4,262	8,416
65,536	-	-	-	-	6,481	9,947

Table 5.2: Time in microseconds for Linux Cluster establish length

As can be seen from the table six different tests were performed: three for the 16 bit libraries, and three for the 32 bit libraries. Looking first at the single processor 16 bit results, it can be seen that the time taken is roughly proportional to the size of the arguments, this is to be expected as the *Approximate CRT Reconstruction* has a sequential time complexity of $O(n)$.

The times taken when using the TTL_PAPERS board, marked as AFAPI, are much larger for the smaller values. There is a communication cost of approxi-

mately 0.6 ms associated with each reduction. Note that this is less for very small values, as less than 4 moduli are needed to reconstruct the number, and hence a full reduction is not required. As the size of the arguments grow, the parallel efficiency increases, with a break even at 4,096 bits and a 2 times speedup at 8,192 bits.

The results for the 16 bit MPI demonstrate that the latency of the Ethernet is too much for any increase in speed, over a single processor. A fixed communication cost of around 4 ms, is just too much, to get any increase in speed.

The times for the 32 bit single processor library are approximately the same as the 16 bit times. The benefit of needing half the number of residues being offset with the increased cost of using 64 bit integers. Note that the maximum size that the single processor can handle is less than the parallel versions, as it has only a quarter of the total memory.

The communication cost of the AFAPI version is approximately doubled to 1.2 ms, when the 32 bit moduli are used. This is because 64 bit, and not 32 bit, numbers are being communicated. The break even point is at 8,192 bits, not 4,096 bits, due to these doubled cost. At 32,768 bits a 2 times speedup is seen, and it can be seen that if the single processor time is extrapolated, then there would be a speedup of 3 times at 65,536 bits.

The MPI libraries are much more suited to the 32 bit moduli. The communication cost is approximately the same at around 4 ms, but with an increase in calculation times for the larger numbers, the libraries break even at 32,768 bits and would have about a 2 times speedup at 65,536 bits.

4 processor Solaris server

Exactly the same tests have been run on the 4 processor Solaris server. This machine has significantly more memory than the Linux box, allowing 256 megabytes of tables to be used with the 32 bit libraries.

The 16 bit single processor results are approximately twice as fast as the Linux results. This is to be expected, as the processors are significantly faster. The Linux machines having 150Mhz Pentium processors and the Solaris machine having 300MHz UltraSparc 2's.

The 16 bit AFAPI results show a much smaller communication cost as would be expected when using shared memory. The communication time is approxi-

Size	16bit			32bit		
(Bits)	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
16	6	12	625	6	17	698
32	8	12	617	10	28	661
64	12	13	572	14	26	642
128	20	17	557	23	30	679
256	38	22	673	39	43	652
512	69	34	588	72	69	729
1,024	131	58	656	135	131	859
2,048	256	108	741	278	296	846
4,096	545	209	1,085	555	760	1,118
8,192	1,106	386	1,127	1,101	1,491	1,752
16,384	-	-	-	2,411	2,352	3,108
32,768	-	-	-	5,494	5,270	5,414
65,536	-	-	-	9,322	8,153	11,365
131,072	-	-	-	19,545	14,259	19,402

Table 5.3: Time in microseconds for Solaris Server establish length

mately 0.01 ms, giving much better parallel efficiency, with a break even at 128 bits and an approximate 3 times speedup at 8,192 bits.

The 16 bit MPI results are also much improved, with a communication time of approximately 0.6 ms. This is approximately the same communication time as the 16 bit Linux AFAPI libraries. However, with the processor being twice as fast, it can only just break even at 8,192 bits.

Looking at the 32 bit results, the single processor results are almost identical to the 16 bit results. The AFAPI and the MPI results are however, a little strange. The AFAPI library breaks even at around 512 bits, but after this point there is no real speedup, a similar thing happens with the MPI libraries.

This lack of performance is, probably, due to the problems of securing all of the processors on a server, which is used by hundreds of users. Although all these tests were performed when the load on the system was minimal, it is likely that as the tests continued, the priority of the processes dropped and they were then swapped out. This would have left the other processes waiting to synchronise, and hence, even the processes running would be unable to continue.

20 processor SGI server

With such disappointing results from the Solaris server, a search was made for a better performing shared memory machine. Some time was secured on a 20 processor machine where it was hoped that 4 processors could be secured for the duration of each test. By waiting until the load dropped to below 15 the results in Table 5.4 were obtained.

Size	16bit			32bit		
(Bits)	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
16	5	33	259	5	30	230
32	7	35	246	9	31	230
64	11	33	253	14	33	220
128	18	33	248	21	34	223
256	32	36	262	35	37	228
512	60	44	258	63	45	243
1,024	115	58	302	119	63	256
2,048	226	87	317	231	88	288
4,096	445	145	373	455	146	355
8,192	906	264	516	904	262	491
16,384	-	-	-	1,864	492	757
32,768	-	-	-	4,170	1,009	1,373
65,536	-	-	-	8,812	2,206	2,666
131,072	-	-	-	20,262	5,412	5,711

Table 5.4: Time in microseconds for SGI Server establish length

As can be seen the problems of the Solaris server were not experienced on the SGI machine. The single processor results are around 20% faster than the Solaris server and again 16 and 32 bit are identical.

The AFAPI results show a communication time of around 0.03 ms, which is 3 times slower than the Solaris AFAPI. However, the results scale perfectly. For the 16 bit libraries there a small speedup at 512 bits, and a best speedup of 3.4 times. The 32 bit libraries also achieve a small speedup at 512 bits and a 4 times speedup at 32,768 and 65,536 bits.

The MPI results are also much better on this machine, with a communication time of around 0.25 ms. This may be, in part, due to the native implementation of the MPI libraries. The gap between the AFAPI and MPI times is between 0.2 and 0.3 ms for all the results, as would be expected.

Maspar MP-2

The results for the Maspar MP-2 are not affected by the size of the number being tested. This can be seen in Table 5.5. The 32 bit results are slower than the 16 bit results, as there is no decrease in work due to the reduction in number length.

Size(Bits)	16bit	32bit
16	1,468	2,098
32	1,495	2,136
64	1,472	2,151
128	1,495	2,136
256	1,480	2,120
512	1,503	2,166
1,024	1,472	2,120
2,048	1,487	2,166
4,096	1,472	2,136
8,192	1,495	2,151
16,384	1,472	2,105
32,768	1,487	2,166
65,536	-	2,121

Table 5.5: Time in microseconds for Maspar establish length

When compared to the other machines, the Maspar is only faster, when the numbers are very large.

5.3.4 Base Extension

To test the base extension code, each power of two had its number of residues doubled. Again, this was repeated until the number of residues needed to store the power of two, equaled the maximum number of residues.

Linux Cluster

The results for the Linux cluster are shown in Table 5.6. Note that the length of the number is shown in the left hand column. The values of the numbers extended are all powers of two. The size listed, is the number of residues being extended to, the original number of residues being half this value.

Len	Size	16bit			32bit		
		SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
2	4	9	68	935	31	122	971
3	16	24	100	1,464	72	336	1,647
5	16	29	286	2,261	98	546	2,651
9	32	63	395	2,308	320	1,057	2,523
17	64	165	803	2,695	1,164	1,552	3,462
33	128	512	1,376	2,519	4,114	3,685	3,952
65	256	1,671	2,917	4,006	14,706	8,463	7,653
129	512	6,054	6,761	4,567	59,035	27,187	23,394
257	1,024	23,377	14,698	10,014	233,224	78,818	71,748
513	2,048	-	-	-	924,684	276,268	310,939
1,025	4,096	-	-	-	-	1,240,824	1,253,870

Table 5.6: Time in microseconds for Linux Cluster Base Extension

The sequential time needed for a base extension should grow quadratically with the length of the number. As the numbers double in size at each step, it would be expected that the time taken, should increase by a factor of four. The communication cost is proportional to the length of the number and so should double in each step.

The 16 bit single processor results show a ratio between each result which is slightly less than four. The 32 bit results are much slower than the 16 bit results, with a difference of about 10 times. This is in part due to the removal of the prime products from the `Inverses32` array.

Looking at the 16 bit results the increased efficiency at passing large messages has meant that the MPI is faster than the AFAPI. The AFAPI almost breaks even when the length is 129, and has a 1.6 times speed increase with numbers of length 257. The MPI, however, more than breaks even at a length of 129, and has a 2.3 times speed increase at 257.

The 32 bit results show better results from the AFAPI, breaking even at a length of 33 and having a 3.3 times speed increase at 513. The MPI has similar results with large numbers, but significantly worse results with small numbers.

4 processor Solaris server

Table 5.7 shows a similar set of results to those obtained when the function **establish_length** was tested. Again a small but non-optimal parallel speedup is obtained, using the 16 bit libraries. The 32 bit libraries, however, show an early speedup for AFAPI at a length of 17, but this is lost at a length of 129, and is only recovered for the largest numbers. The 32 bit MPI fails to achieve any speedup at all.

Len	Size	16bit			32bit		
		SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
2	4	6	12	222	15	22	263
3	16	14	20	288	28	37	331
5	16	15	26	350	44	62	405
9	32	27	41	448	136	139	556
17	64	69	70	629	447	399	1,502
33	128	201	111	759	1,681	1,518	2,407
65	256	675	354	888	6,828	5,804	7,265
129	512	2,313	1,210	2,265	25,992	43,526	32,287
257	1,024	8,872	4,628	7,781	104,512	133,354	155,557
513	2,048	-	-	-	438,926	380,037	492,489
1,025	4,096	-	-	-	1,670,612	1,870,774	2,066,018
2,049	8,192	-	-	-	6,710,963	5,568,338	5,952,631

Table 5.7: Time in microseconds for Solaris Server Base Extension

20 processor SGI server

Compared to the Solaris server, better results are obtained by the 20 processor machine, as can be seen in Table 5.8.

As was the case with both the previous machines there is a huge difference in the times for the 16 and 32 bit base extensions. Again the gaps between consecutive results tends towards four as would be expected.

The 16 bit AFAPI libraries give good results with a four times speedup at a length of 257, and a break even as low as 17. The MPI does not do quite as well, with a speedup of 3.5 times at 257, and a break even at a length of 65.

The 32 bit results are much better, mainly due, to the increased problem size. The AFAPI manages to break even at a length of 9, have a 2 times speedup at a

Len	Size	16bit			32bit		
		SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
2	4	5	27	75	14	34	85
3	16	11	42	102	27	50	120
5	16	12	54	131	42	66	152
9	32	22	57	136	130	90	183
17	64	52	66	144	427	161	265
33	128	142	84	167	1,600	440	622
65	256	448	160	250	6,319	1,558	2,112
129	512	1,532	436	566	24,627	5,919	7,576
257	1,024	5,637	1,466	1,629	98,033	23,543	29,896
513	2,048	-			391,949	93,583	119,784
1,025	4,096	-			1,564,033	375,733	450,021
2,049	8,192	-			6,237,853	1,507,162	1,788,481

Table 5.8: Time in microseconds for SGI Server Base Extension

length of 17, and achieves full speedup by a length of 65. The speedup is in fact super-linear beyond this point, probably due to caching effects.

The 32 bit MPI results are also good, with a break even at 17, and a best speedup of 3.5 times.

Maspar MP-2

There are no results for the Maspar as it does not need to perform base extensions.

5.3.5 Reconstruction to a small modulus

The last of our basic tests is the reconstruction to a small modulus. In this test the power of two is reconstructed to a 32 bit modulus for the 16 bit library, and a 64 bit modulus for the 32 bit library.

Linux Cluster

These results should be similar to those for the function `establish_length`. An increase in calculation time is expected however as an inverse will need to be calculated for each residue. The results are shown in Table 5.9.

The single processor results are as expected, with an execution time which becomes proportional to the number size as the length increases. The 32 bit

Size	16bit			32bit		
(Bits)	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
32	23	424	3,836	54	565	5,040
64	39	499	3,810	83	716	4,579
128	70	600	3,890	139	921	4,614
256	130	761	4,208	254	1,259	5,525
512	247	800	4,028	486	1,200	4,711
1,024	468	822	4,722	939	1,577	6,246
2,048	923	1,062	5,220	1,840	2,268	6,064
4,096	1,859	1,101	4,401	3,655	2,061	5,560
8,192	3,703	1,564	4,782	7,213	3,171	7,136
16,384	-	-	-	14,540	5,089	8,329
32,768	-	-	-	29,121	9,507	15,271
65,536	-	-	-	-	16,217	25,047

Table 5.9: Time in microseconds for Linux Cluster modCRT_32u/modCRT_64u

results are just over twice those of the 16 bit results which is due to the increased time in calculating the inverses.

The AFAPI results show a communication time of approximately, 0.6 ms for the 16 bit libraries and 1.2 ms for the 32 bit libraries. These costs are exactly the same as for the establish length. The MPI libraries have a communication time of approximately 3.8 ms for the 16 bit libraries and 5 ms for the 32 bit libraries.

The 16 bit AFAPI libraries manage to obtain a speedup at 4,096 bits and an ultimate speedup of 2.4 times. The MPI does not achieve any speedup.

The 32 bit AFAPI does much better, with a 1.5 times speedup at 4,096 bits, a 3 times speedup at 32,768, and a projected speedup of 3.5 times at 65,536 bits. With the increased problem size, the MPI manages to break even at 8,192 bits and has a projected speedup of 2.3 times at 65,536 bits.

4 processor Solaris server

The Solaris server repeats its results for the last two tests by, achieving a small speedup with 16 bit AFAPI, and no speedup using the 32 bit libraries. This can be seen in Table 5.10.

Size (Bits)	16bit			32bit		
	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
32	10	28	523	20	98	704
64	17	34	545	30	132	721
128	31	41	598	51	150	828
256	57	55	565	94	209	1,640
512	110	90	791	178	298	1,542
1,024	208	143	732	343	464	1,616
2,048	411	253	967	686	846	1,515
4,096	827	484	1,108	1,356	1,574	2,384
8,192	1,665	965	1,808	2,688	2,882	4,104
16,384	-	-	-	5,412	6,142	8,693
32,768	-	-	-	10,847	11,726	12,537
65,536	-	-	-	21,694	24,235	32,532
131,072	-	-	-	43,805	45,532	46,661

Table 5.10: Time in microseconds for Solaris Server modCRT_32u/modCRT_64u

20 processor SGI server

The results for the SGI server can be seen in Table 5.11.

The results for the modular reconstruction are very similar to those for the establishing length. Both the AFAPI and the MPI, achieve good speedup in both 16 bit and 32 bit versions. The higher latency of the MPI means that larger numbers are needed to achieve speedup, than with the AFAPI. They both achieve almost 4 times speedup with the larger numbers, when using the 32 bit library.

Maspar MP-2

The results for the Maspar are shown in Table 5.12.

The results for the modular reconstruction differ significantly from those for establishing length. Instead of being completely constant, the time taken does increase slightly with problem size. This is due to the time taken calculating inverses. The time taken will be depend on the number of iterations needed to perform the extended Euclidean algorithm. The greater the number of inverses, the greater the chance, that at least one of them, will require a lot of iterations.

The difference between the 32 bit and the 16 bit results, comes down to the processors. The Maspar MP-2 has 4 bit processors [28], these are very slow when

Size (Bits)	16bit			32bit		
	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
32	11	59	183	22	111	239
64	19	62	183	33	117	267
128	35	66	187	56	124	245
256	65	73	200	104	131	260
512	124	90	224	199	168	288
1,024	233	122	242	380	215	333
2,048	457	179	296	741	311	437
4,096	906	296	410	1,498	520	650
8,192	1,796	532	640	2,963	908	1,024
16,384	-	-	-	5,984	1,660	1,827
32,768	-	-	-	11,997	3,239	3,407
65,536	-	-	-	24,035	6,479	6,550
131,072	-	-	-	48,212	13,317	13,044

Table 5.11: Time in microseconds for SGI Server modCRT_32u/modCRT_64u

Size(Bits)	16bit	32bit
32	2,281	18,459
64	2,288	18,799
128	2,303	19,407
256	2,273	19,409
512	2,303	20,874
1,024	2,273	21,605
2,048	2,273	21,973
4,096	2,502	22,705
8,192	2,670	23,315
16,384	2,670	24,534
32,768	2,639	25,024
65,536	-	25,268

Table 5.12: Time in microseconds for Maspar modCRT_32u/modCRT_64u

performing 64 bit divisions. The timings of all basic arithmetic functions are given in [29], it takes 90 clock ticks to perform a 32 bit division, and 280 clock ticks to perform a 64 bit division.

5.4 Timing results for basic arithmetic

In this section, the basic arithmetic operations of multiplication and division are timed, and then an example of calculating Fibonacci numbers is given. The purpose of these tests is both to demonstrate the correctness of the libraries by giving it larger tasks to perform, and also to measure the time taken by the operations. For these tests, timing results for the shared memory machines have not been given, due to the problem of getting a low enough system load for consistent results

5.4.1 Multiplication

To test the correctness of multiplication, factorials were calculated, up to the limit of the number range. The results were compared with outputs from other programs, and seen to be correct. For the timing test however, two large numbers are multiplied together.

To split up the factorial into two roughly equal halves, the smallest factorial greater than a power of two is calculated. The calculation then continues, but using a new product, until that, too, is larger than the power of two. When multiplied together, the two halves form one large factorial.

Linux Cluster

The results for the Linux machines is given in Table 5.13. Note that the times given do not include the cost of any base extensions.

The results are consistent with the results for establishing length. This is to be expected, as the time taken by multiplication is dominated by the time taken to establish the length of the number. A small increase in both communication and total times, are due to the inaccuracy in estimating the length of the result of multiplication. As in the other tests the 16 bit AFAPI only achieves a speedup

Size	16bit			32bit		
(Bits)	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
32	20	653	6,143	34	1,334	6,692
64	34	981	6,371	37	1,168	5,919
128	67	1,250	5,987	64	2,217	6,493
256	119	1,172	5,319	118	2,140	5,383
512	212	1,004	5,872	219	2,520	7,337
1,024	470	1,502	8,114	467	2,041	6,471
2,048	836	1,296	8,356	919	3,392	8,652
4,096	1,595	1,954	5,723	1,895	2,349	6,009
8,192	4,144	2,339	6,307	4,594	3,785	9,838
16,384	-	-	-	7,763	4,414	8,832
32,768	-	-	-	14,222	6,229	10,936
65,536	-	-	-	-	11,619	17,582

Table 5.13: Time in microseconds for Linux Cluster multiplication

with the larger numbers, and the both the MPI and the AFAPI can achieve reasonable speedup, with the 32 bit libraries.

Maspar MP-2

The results for the Maspar are shown in Table 5.14.

The results for the Maspar are very similar to those for establishing length. The 32 bit results are slightly larger than the 16 bit results as before. However the last 16 bit result is of interest.

Using 1024 16 bit moduli it was possible to predict the length of a multiplication with an error of, at most, one. However with 4096 moduli, the potential value of δ, that is the number of extra moduli needed, grows to over a hundred. It is not possible to predict this value as accurately, and hence more iterations are needed when more the numbers have lengths above 1024.

5.4.2 Division

To test the correctness of division, a Euclidean GCD was calculated, using a range of values. The code for this test can be seen in Appendix B.4.1. The code demonstrates the basic workings of the library including, intialisation, input,

Size(Bits)	16bit	32bit
32	2,151	3,570
64	2,166	3,128
128	2,273	3,219
256	2,273	3,219
512	2,166	3,128
1,024	2,533	3,265
2,048	2,212	3,189
4,096	2,151	3,143
8,192	2,532	3,692
16,384	2,776	3,112
32,768	4,943	3,173
65,536	-	3,631

Table 5.14: Time in microseconds for Maspar multiplication

timing and basic arithmetic. Calculating GCDs in this way involves thousands of divisions, and as such, is likely to find any errors in the division functions.

To time the division function, the numbers produced in the multiplication example were used. The multiplication was reversed using the division function. This should result in a zero remainder.

Linux Cluster

The results for the Linux cluster can be seen in Table 5.15. The size quoted is the size of the number before division, with the divisor being approximately half the size of dividend. Note that these results are in milliseconds, not microseconds.

Looking at the single processor results, it is clear that the cost of division increases quadratically with the size of the number.

The parallel performance is not as good, as it was for multiplication. The AFAPI libraries manage some speedup with the 16 bit libraries, and a reasonable speedup of 2 times, at 32,768 bits, using the 32 bit library.

Maspar MP-2

The results for the Maspar can be seen in Table 5.16.

As can be seen in these tables, the time taken to perform division on the Maspar, is proportional to the size of the arguments. However unlike multiplication

Size	16bit			32bit		
(Bits)	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
32	0.4	9	117	0.8	15	121
64	0.8	18	157	1	15	152
128	2	41	285	2	32	183
256	5	73	483	4	64	307
512	18	170	929	13	164	535
1,024	72	288	1,985	38	260	849
2,048	253	668	3,584	157	592	1,732
4,096	926	1,536	7,012	644	1,144	3,835
8,192	4,081	3,521	14,319	2,287	2,638	8,095
16,384	-	-	-	9,628	6,977	16,779
32,768	-	-	-	39,418	19,956	39,684
65,536	-	-	-	-	58,962	102,548

Table 5.15: Time in milliseconds for Linux Cluster division

Size(Bits)	16bit	32bit
32	44	97
64	67	98
128	110	146
256	196	238
512	373	432
1,024	729	809
2,048	1,410	1,598
4,096	2,852	3,176
8,192	5,672	6,344
16,384	11,137	12,695
32,768	22,273	25,980
65,536	-	51,734

Table 5.16: Time in milliseconds for Maspar division

the Maspar is no faster than the Linux cluster at division even using the largest numbers.

5.4.3 Calculating Fibonacci numbers

The time taken to perform an addition is much the same as the time taken to perform a multiplication. To test the libraries more completely Fibonacci numbers were calculated using addition. This is not intended to be a high speed calculation of the Fibonacci number but, a test of addition and subtraction.

In each test the nth Fibonacci number is calculated by addition. The process is then reversed using subtraction. Like calculating GCDs, this involves a lot of steps, and so is likely to expose errors in the libraries. Listed below are the times taken to perform the addition, the subtraction being around 25% quicker as no base extensions are required.

Linux Cluster

The results for the Linux cluster can be seen in Table 5.17, note n represents the nth Fibonacci number.

n	16bit			32bit		
	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
1000	130	1,079	6,378	169	2,102	6,228
2000	643	2,340	11,640	665	4,573	12,455
4000	2,233	5,182	24,240	2,582	9,703	25,990
8000	9,075	11,887	50,687	9,649	20,160	51,431
16000	35,924	27,975	107,844	37,791	46,032	107,473
32000	-	-	-	151,844	113,139	240,755
64000	-	-	-	615,762	312,245	590,144

Table 5.17: Time in milliseconds for Linux Cluster Fibonacci

The time taken should grow quadratically with n, both because of the additions, and because of the base extensions. With a single processor, this is seen to be the case.

There is little difference between the 16 and 32 bit results, which suggests that it is the cost of the additions, not the base extensions that is significant.

In parallel, speedups are similar to those achieved for division, with a best increase of 2 times for the 32 bit AFAPI. The MPI can only achieve a slight speedup, with the largest numbers, using the 32 bit library.

Maspar MP-2

The results for the Maspar can be seen in Table 5.18. Note that there are no 32 bit results because the Maspar MP-2 has developed a hardware fault, and is not operational at the time of writing.

n	16bit	32bit
1000	2,160	-
2000	4,308	-
4000	8,609	-
8000	17,207	-
16000	34,410	-
32000	68,792	-
64000	137,593	-

Table 5.18: Time in milliseconds for Maspar Fibonacci

The time taken is, not surprisingly, directly proportional to the number of additions needed. This is because each addition takes a constant amount of time. There is also no base extension though this would also take time proportional to the lengths of the numbers.

Above 32,000 the Maspar results are better than the Linux 32 bit AFAPI results, though it is likely that better results could be achieved using the 20 processor SGI server.

5.5 Conclusions

In this chapter six different timing tests have been described. The first three tests looked at the underlying functions which are called repeatedly during arithmetic. These were `establish_length`, `base_extension` and `modCRT_32u/64u`.

In these tests it was clear that the TTL_PAPERS circuit board with the AFAPI libraries, was much better suited to these libraries, than Ethernet using

the MPI libraries. What was more surprising, was the large differences in communication times of the shared memory AFAPI and MPI results. I would assume that this is due to the large level of error checking used by the MPI libraries, something that is lacking from the AFAPI libraries.

In terms of parallel speedup the TTL_PAPERS based AFAPI results showed consistently, that a small increase of about 2 times could be achieved using the 16 bit libraries when the largest numbers were used. However, speedup only began with numbers of half the maximum size, giving it a small useful range.

Using 32 bit moduli both the AFAPI and MPI were capable of speedup, with the AFAPI achieving between 3 and 4 times speedup in all tests. Speedup was only achieved at the high end of the number range, but if the 32 bit libraries are needed, then the AFAPI libraries will give a speedup.

Comparing the 16 and 32 bit results showed the 16 bit to be slightly faster in establishing length and reconstructing to a small modulus. Base extension however was around 10 times slower using the 32 bit libraries. Overall using the 16 bit libraries is always preferable unless you need the range of the 32 bit libraries.

The 4 processor Solaris server gave terrible results, this was probably due to the problems associated with processes being swapped on and off the processors. The 20 processor SGI server gave excellent results however, with parallel efficiencies approaching 100% in all the 32 bit tests. More importantly, speedup was achieved with numbers of a few hundred bits, which are very small compared to the numbers the library could handle. Given exclusive use of such a shared memory machine the libraries could give excellent parallel speedup for a wide range of numbers, using many more than four processors.

The Maspar is at the very limit of parallelism possible for this style of calculation. By using one processor per modulus all the complexities drop a complexity class. With $O(n^2)$ sequential operations becoming $O(n \log n)$, and $O(n)$ operations becoming $O(\log n)$. However, as the reduction operations are fixed in size the results appear to be $O(n)$ and $O(1)$. Unfortunately the Maspar is a very old machine, and there are not many left that are working.

The arithmetic tests showed that the basic functions do dominate the time taken for calculations, with the results mirroring those found previously.

Overall, the fastest results were found using a shared memory machine, which

gave good parallel efficiency. The Maspar shows perfect scaling with number size but has to be using very large numbers to gain any speedup over more modern workstations. AFAPI was faster than MPI in all cases, and the Linux Cluster gives an interesting range of results not having the same problems as the 4 processor Solaris server.

Chapter 6

Decreasing the cost of division

6.1 Introduction

It takes significantly longer to perform a general division, than it takes to perform any of the other arithmetic operators. In this chapter we will look at two different methods of avoiding general division.

The first method, is to check if the division is exact, in which case it is possible to perform the division by multiplying each residue by the inverse of the divisor. This takes an amount of time comparable with multiplication and is one of the main advantages of using a modular representation.

To know if a division is exact, a divisibility test can be used, however deterministic divisibility tests, have similar complexities to division itself. For this reason, a probabilistic test is used, which can give a probable answer with the same complexity as multiplication.

If the division is not exact, all is not lost. It may be that you have several divisions to make using the same divisor. In subsection 1.3.4 three different methods were described, each of which allowed a division by a known divisor to be performed.

The method chosen for implementation, is one of the methods proposed in [22], in which the value $\lfloor M/B \rfloor$ is pre-calculated. To use this technique, a reverse base extension is required, which allows a division by the product of moduli, M.

6.2 Exact division and a divisibility test

6.2.1 Exact division

The function for exact division was described in Section 3.12. In most cases it can be performed with a sequential time complexity of $O(n)$, and a parallel time complexity of $O(n/nproc) + O(\log nproc)$.

This is much better than general division, which has a sequential complexity of $O((n-m)n)$ and a parallel complexity of $O((n-m)n/nproc) + O((n-m)\log nproc)$, without base extension, where n is the length of the dividend, and m is the length of the divisor.

That is to say, that a general division, has the same time complexity, as $n-m$ exact divisions.

It will be worth performing a divisibility test if the likelihood of an exact division is high and the cost of a divisibility test is low. The first of these conditions will depend on the end use of the libraries, the second will be addressed below.

6.2.2 Deterministic divisibility tests

If at all possible it is preferable to have a deterministic, rather than a probabilistic algorithm. However in many cases, a probabilistic algorithm offers a significant decrease in complexity.

While a proof of the minimum complexity of a deterministic divisibility test is not going to be given, an example of a possible test will highlight some of the problems encountered.

Given two numbers A and B both of which have lengths between n and $2n$, then it will be possible to calculate $C \equiv A * B^{-1} \bmod p_i$, where $0 \leq i < 2n$. This will take a time proportional to $2n$.

If B divides A, then C will equal A/B and will be less than $M(2n)/B$. However if B does not divide A, then C will be greater than $M(2n)/B$. To see that this is true, consider a value of C, which is congruent to A/B modulo $M(2n)$. It will obey Equation 6.1.

$$B.C = A + kM(2n) \tag{6.1}$$

If B divides A, then $k = 0$ and $C = A/B < M/B$. If B does not divide A, then $k \geq 1$ and $C = A/B + kM/B > M/B$.

If the length of C can be calculated, it will then be possible to determine if B divides A. However all the deterministic methods used to calculate length in Section 2.3, have sequential time complexities, which are at least the square of the number's length.

Described in Section 2.3 is a probabilistic length test which has a sequential complexity of $O(n \log n)$, but as will be shown in the next section, this can be improved upon.

6.2.3 A probabilistic divisibility test

In this subsection a new divisibility test is proposed. By using a probabilistic confirmation function the time complexity can be significantly reduced.

To perform the probabilistic divisibility test an exact division is again performed the result of which is C. If B divides A then the division will work and $BC = A$, if B does not divide A, then there does not exist a value of C such that $BC = A$.

The test checks whether $BC \equiv A \bmod R_i$, where R_i is the i^{th} random number chosen. It is important that the random numbers chosen are not divisible by any of the moduli, this can be checked in time proportional to the number of moduli, though it would be quicker to choose random numbers which are less than the smallest modulus when using 32 bit moduli.

For each test, the numbers A,B and C are reconstructed, using the function `modCRT_32u`, which performs the *Reconstruction to a Small Modulus* in a time of $O(n/nproc) + O(\log nproc)$.

If the congruence is found not to hold then it is certain that the numbers do not divide, but if the congruence does hold then it is more probable that the numbers do divide.

The chance that two numbers will be congruent to an arbitrary modulus is equal to $1/R_i$. However, there may be common factors in the moduli chosen, so the chance that two non-equal numbers will pass s tests, is $1/lcm(R_0, ..., R_{s-1})$. As was shown in subsection 4.5.4, the chance of passing 10 tests, using the function `random` to generate the numbers, is 2^{-289}.

Using a fixed number of confirmation steps means that the overall complexity of this test is $O(n/nproc) + O(\log nproc)$ plus the cost of an exact division.

6.2.4 Timing Results

Table 6.1 contains the timing results for the divisibility test. Numbers of the size shown were tested for divisibility, by numbers half their size. The 32 bit Linux libraries were used for all tests. Note that this table is in microseconds.

Size	Divides			Does Not Divide		
(Bits)	SINGLE	AFAPI	MPI	SINGLE	AFAPI	MPI
32	612	12,268	140,994	146	2,659	20,422
64	888	12,841	129,663	222	2,407	16,639
128	1,572	19,631	156,614	244	2,989	22,737
256	2,698	19,942	136,448	528	3,620	21,484
512	4,638	20,254	126,909	705	3,333	23,622
1,024	9,664	25,727	138,681	1,344	3,541	21,071
2,048	17,389	24,584	136,745	2,601	4,418	24,700
4,096	34,488	43,307	127,204	5,476	5,603	19,534
8,192	71,133	38,331	135,814	10,286	6,019	19,242
16,384	137,659	68,686	155,961	21,753	10,294	31,969
32,768	268,877	99,947	231,887	44,805	15,812	30,239
65,536	-	169,431	289,759	-	24,430	43,390

Table 6.1: Time in microseconds for Linux Cluster 32 bit divisibility test

The cost of the divisibility test on a single processor is proportional to the size of the inputs. There is a big difference between the case when the numbers are divisible and when they are not. This is expected, as it takes ten confirmation steps to be passed as dividing, but non-divisors are likely to be detected in the first step. The ratio between the two sets of results, is not ten however, as the exact division will take place before the confirmation steps.

The parallel results follow the same pattern as before, the AFAPI library achieving some speedup at 8,192 bits, and have a 3 times speedup at 32,768 bits, when the number do not divide.

The MPI libraries only just manage a speedup, which is consistent with the results for the *Reconstruction to a Small Modulus.*

Tests on other platforms, also give results which are consistent with the result for *Reconstruction to a Small Modulus.*

In general the results do not seem to increase as smoothly as in other tests. This is thought to be due to other processes being run at the same time as the test. In particular the 4,096 bit AFAPI result for the case when the numbers divide seems significantly higher than it should be.

6.2.5 Conclusion

We have seen that a probabilistic divisibility test is capable of testing for exact division in time that grows linearly with size of the arguments. The fastest 32 bit Linux division takes 15 ms, but a successful divisibility test takes only 0.6 ms. At the other end of the scale a 65,536 bit division takes 59 seconds using the AFAPI libraries compared with 0.17 seconds for a successful divisibility test or 0.024 seconds for a failed one.

If there is a reasonable possibility of a division being exact, it would be worth testing for divisibility, before proceeding with a general division.

6.3 Division by a known divisor

In some situations you will always be dividing by the same number, one example of this is performing modular arithmetic with large moduli, such as during RSA encryption/decryption.

An implementation of a method suggested in [22], is described, an integral part of which, is a division by the product of moduli M.

6.3.1 Hung's known division method

If the value of $\lfloor M/B \rfloor$ is known, then it is possible to produce a number which is approximately M times larger than $\lfloor A/B \rfloor$, by multiplying $\lfloor M/B \rfloor$, by A. If the result of this multiplication is then divided by M, a number close to $\lfloor A/B \rfloor$ will result. The error in this calculation is shown in Equations 6.2 and 6.3.

$$A\left\lfloor \frac{M}{B} \right\rfloor = M\left(\left\lfloor \frac{A}{B} \right\rfloor + \frac{|A|_B}{B} - \frac{A|M|_B}{BM}\right) \tag{6.2}$$

$$\left\lfloor \frac{A\left\lfloor \frac{M}{B} \right\rfloor}{M} \right\rfloor - \left\lfloor \frac{A}{B} \right\rfloor = \left\lfloor \frac{|A|_B}{B} - \frac{A|M|_B}{BM} \right\rfloor \tag{6.3}$$

The size of the error will be dependent on the exact values of A and B, and also the value of M chosen. It is certain that the term $\frac{|A|_B}{B}$ will be less than one as $|A|_B < B$ by the definition of $|A|_B$. The second term will depend on the relative sizes of A and M, by setting M to be larger than A the second term will also be less than one.

With $M > A$, the error can either be, 0 or -1, as a positive error would be lost in truncation. This leaves the problem of how to perform a division by M, which is described next.

6.3.2 Exact division by M

Before looking at general division by M, first an exact division is described, which will be used by the general division.

If M is the product of $len1$ moduli, then a multiple of this number will have a length of $len2$, where $len2 > len1$. The two lengths cannot be equal, as it will take $len1 + 1$ moduli to store M.

To perform the exact division, each of the residues x_i where $size1 \leq i < size2$ is multiplied by $M^{-1} \bmod p_i$. After this multiplication there will be $size2 - size1$ moduli which are storing the correct value of the exact division.

To return the number to a standard form, it is necessary to reconstruct the values for at least the first $size2 - size1$ moduli. (The number of residues used is always of the form $2^n nproc$ so it is likely that more residues are required). The reconstruction of these moduli is a reverse base extension.

6.3.3 Reverse base extension

In a standard base extension the number of residues is increased by using the first len moduli to reconstruct subsequent moduli. The method for performing this task is described in Section 2.4 and Section 3.8.

The method used is, first, to calculate the reconstruction constants, y_i, for len moduli. The value of y_i is defined as:

$$y_i = \left(\left(\frac{\prod_{j=0}^{len-1} p_j}{p_i} \right)^{-1} x_i \right) \bmod p_i. \tag{6.4}$$

To calculate the new residues x_j, Equation 6.5 is used.

$$x_j = \left(\prod_{i=0}^{len-1} p_i \right) \left(\left(\sum_{i=0}^{len-1} p_i^{-1} y_i \right) - k \right) \bmod p_j \tag{6.5}$$

To change this to a reverse base extension, the two equations become Equation 6.6 and Equation 6.7.

$$y_i = \left(\left(\frac{\prod_{j=size1}^{size2-1} p_j}{p_i} \right)^{-1} x_i \right) \bmod p_i. \tag{6.6}$$

$$x_j = \left(\prod_{i=len1}^{len2-1} p_i \right) \left(\left(\sum_{i=size1}^{size2-1} p_i^{-1} y_i \right) - k \right) \bmod p_j \tag{6.7}$$

This looks like a very small change, however, for a normal 16 bit base extension, the values of $\left(\frac{\prod_{j=0}^{len-1} p_j}{p_i} \right)^{-1} \bmod p_i$, $p_i^{-1} \bmod p_j$ and $\left(\prod_{i=0}^{len-1} p_i \right) \bmod p_j$ are all stored in tables. The value of k will have been set when `establish_length` is called.

For a 32 bit base extension the values of $\left(\prod_{i=0}^{len-1} p_i \right) \bmod p_j$ have to be calculated, otherwise it is the same as for the 16 bit case.

For a reverse base extension, the value of $\left(\frac{\prod_{j=size1}^{size2-1} p_j}{p_i} \right)^{-1} \bmod p_i$, can be calculated by multiplying $\left(\frac{\prod_{j=0}^{size2-1} p_j}{p_i} \right)^{-1} \bmod p_i$, which is stored in `Inverses16`, by $\left(\prod_{i=0}^{len1-1} p_i \right) \bmod p_j$.

In the 16 bit case the value of $\left(\prod_{i=0}^{len1-1} p_i \right) \bmod p_j$ is stored in `Inverses16`, but as was the case for a standard base extension, this product will have to be calculated in the 32 bit case.

For the 16 bit reverse base extension, the values of $p_i^{-1} \bmod p_j$ can be read from the array `p_inverse`, but for the 32 bit version the function `inverse` needs to be used, as only the values of $p_i^{-1} \bmod p_j$ where $i < j$, are stored.

The value of $\left(\prod_{j=len1}^{len2-1} p_i \right) \bmod p_j$, can be calculated using the values stored

in **Inverses16** as is shown in Equation 6.8. Note this does not work for a normal base extension where the value of j is greater than $len - 1$.

$$\frac{\texttt{Inverses16[j][len1} - 1\texttt{]}}{\texttt{Inverses16[j][len2} - 1\texttt{]}} \equiv \frac{\left(\frac{\prod_{i=0}^{len1-1} p_i}{p_j}\right)^{-1}}{\left(\frac{\prod_{i=0}^{len2-1} p_i}{p_j}\right)^{-1}} \equiv \left(\prod_{i=len1}^{len2-1} p_i\right) \bmod p_j \qquad (6.8)$$

This leaves the value of k, which can be calculated using the new values of y_i. The method used is the same as was described in subsection 2.2.5 and subsection 3.6.2.

6.3.4 General division by M

Described above is a method of performing an exact division by M. Hung's method of known division requires a general division by M. To turn a general division into an exact one, it is necessary to first subtract any remainder.

The reminder after division by M is simply the value stored in the $size1$ moduli, which make up M. This value shall be referred to as R. Using the normal style of base extension, R can be extended so the values of R_i are known up to $R_{size2-1}$. (Again due to the method used to store numbers the number will be extended to a value which is $2^n nproc$).

By subtracting R from the number being divided, the exact division can then take place. Note however that unlike a normal base extension it could be the case that, R almost completely fills its $size1$ residues. In this case the value of k calculated during the base extension may be one too much. In this case R will be extended to $R - M$, and the result of the exact division will be one too many.

As it is known when this error could have occurred, the result of the exact division can be checked by multiplying the returned answer by M. If the product of this multiplication is greater than the original number then an error has occurred, and one is subtracted from the answer.

6.3.5 Implementation of known division

The implementation of the known division is split up into two parts. The first part is to pre-calculate the values of M and $\lfloor M/B \rfloor$. A value of M is chosen that

it is certain to be larger than any value of A used.

To perform a division, the values of A, $\lfloor M/B \rfloor$ and M are passed to the known division function. This in turn multiplies A and $\lfloor M/B \rfloor$ together and then passes the result to a division by M function.

The value returned by the division by M function is always correct, but it may not be equal to $\lfloor A/B \rfloor$. The value may equal $\lfloor A/B \rfloor - 1$. To check if this is the case, the assumed value of $\lfloor A/B \rfloor$ is multiplied by B, and is then subtracted from A. This should yield the remainder after division, which may also be returned if desired. If an error has occurred then the remainder will be greater or equal to B. This can then be corrected by subtracting B from the remainder and adding one to the quotient. As shown above, the error cannot be greater than one, as long as $M > A$.

6.3.6 Results

To test the known division function the test program for division was extended. The time taken calculating $\lfloor M/B \rfloor$ was ignored with only the known division section being timed.

Linux Cluster

The results for the Linux Cluster are shown in Table 6.2, it should be noted that The column labelled G, is the time taken to perform a general division on a single processor. The letter S,A and M, represent SINGLE, AFAPI and MPI.

It is clear from the results that known division is significantly faster than general division, it is however much slower than a normal base extension which can be seen by referring back to Table 5.6. The reason for the lack of performance, when compared with a normal base extension, is the extra work needed calculating constants, which would be normally be read from tables. This was seen with the standard base extension where the lack of the 32 bit `p_inverse` table resulted in a ten fold increase in time, when compared with the 16 bit results.

Looking only at the single processor results, the largest difference between general and known division is 15 times, and the smallest is 2.2 times. The 16 bit results being better, as the `p_inverse` table can be used in both the standard and reverse base extensions.

Size (Bits)	16 bit				32 bit			
	G	S	A	M	G	S	A	M
32	0.4	0.18	3.7	43	0.8	0.34	4.4	43
64	0.8	0.29	6.2	44	1	0.39	7.4	47
128	2	0.50	9.0	59	2	0.58	11	57
256	5	1.7	13	100	4	1.2	28	50
512	18	4.2	20	125	13	2.9	30	56
1,024	72	5.9	24	136	38	8.5	27	73
2,048	253	18	38	164	157	28	51	97
4,096	926	64	71	289	644	161	168	320
8,192	4,081	267	194	457	2,287	395	241	328
16,384	-	-	-	-	9,628	1,220	664	936
32,768	-	-	-	-	39,418	5,161	1,794	2,265
65,536	-	-	-	-	-	-	6,456	7,604

Table 6.2: Time in milliseconds for Linux Cluster known division

Looking at the parallel results, the MPI libraries fail to give any speedup with the 16 bit moduli. They do, however, give a much but better set of results with 32 bit moduli. The AFAPI only just breaks even with 16 bit moduli, but shows a 2.9 times speedup at 32,768 bits. The efficiency of the 32 bit results is helped by the lack of tables.

Maspar MP-2

The results for the Maspar MP-2 are shown in Table 6.3, again the results for both general and known division are shown.

The known division is faster than the general division for all sizes of input both for 16 and 32 bit moduli. This was also seen with the Linux results. With larger numbers the speedup is quite large being 9 times in the best case.

A contrast is seen between the rates of increase in time, with number length. The general division results, shows an exact doubling of time, between all results above 1024 bits.

The known division, however, is almost constant in time for the smaller numbers. This is probably due to the multiplications, subtractions and comparisons needed by the known division. Still, even for the smallest numbers, there is significant advantage when using known division with a worst case speedup of 2

Size	16 bit		32 bit	
(Bits)	GENERAL	KNOWN	GENERAL	KNOWN
32	44	22	97	27
64	67	28	98	31
128	110	62	146	38
256	196	96	238	48
512	373	164	432	71
1,024	729	154	809	116
2,048	1,410	294	1,598	206
4,096	2,852	587	3,176	789
8,192	5,672	1,176	6,344	762
16,384	11,137	4,585	12,695	1,511
32,768	22,273	4,914	25,980	3,016
65,536	-	-	51,734	6,058

Table 6.3: Time in milliseconds for Maspar known division

times.

6.4 Conclusions

In this chapter, two methods of avoiding a general division have been described. A new probabilistic divisibility test can quickly determine if a division is exact. If an exact division can be performed, it could be hundreds of times faster than a general division. If the division is not exact, the time taken by the test is reduced by around 5 times, decreasing the cost of failed tests.

A good time saving can also be achieved if the divisor is known in advance. In the best cases, a division by a known divisor can be around 10 times faster than a general division. This is the first implementation of a known division division algorithm for a modular representation.

It has been seen, that a lack of tables, decreases the efficiency of the base extensions required by the known division. It may be possible to further increase the speed of the known division, by storing all the constants needed for the base extensions. This has not been implemented, but the evidence of the 16 bit verses the 32 bit standard base extension times, suggested it could have a significant effect.

Chapter 7

Examples

7.1 Introduction

While the previous chapters have concentrated on the functions that make up the library. In this chapter, examples are given of the library in use. Two examples are given, calculating GCDs, and calculating determinants.

Four different methods of calculating GCDs are compared. One method is the Euclidean algorithm. This gives poor results, as it is based on general division. The other three methods, all give improvements over the Euclidean algorithm.

Silver's GCD algorithm, also known as the binary GCD algorithm, involves only subtraction, comparison, and bit shifting. It normally gives good results when using a traditional representation.

Lehmer's GCD algorithm avoids the costs of handling large numbers by performing calculations to a limited precision. It uses the most significant bits of a number, in the same way that a general division does.

Weber's GCD algorithm attacks the problem from the other end, by using the least significant bits to perform calculations. Again, by performing calculations to a limited precision, time can be saved.

Determinant calculations are well suited to modular computations, as the determinant can be built up using multiplications, additions and subtractions. There is no need to perform comparisons or general division.

Two implementations are described, one of which uses the standard datatype `t_PMA`, and the other that uses the `t_LPMA` datatype, that was described in Chapter 4. Hadamard's bound is used to calculate the possible size of the determinant,

allowing a fixed number of residues, in the second implementation.

7.2 GCD

As mentioned in the introduction, four different GCD algorithms are compared in this section. Tests results are given both for the Linux Cluster and the Maspar MP-2. As will be seen, the best algorithm is dependent on the platform used.

7.2.1 Euclidean GCD

The code for the Euclidean GCD function can be seen in Appendix B.4.1. Often described as the first algorithm, it should be familiar to almost everyone.

The implementation tested, uses the general division function `r_div` to give the remainder after division. The pair of numbers are reduced, until one is zero. The function `iszero` is used to test if a number is zero. As can be seen in the header files shown in Appendix B.2, `iszero` is a simple macro.

7.2.2 Silver/Binary GCD

Silver's GCD algorithm is well suited to even the smallest of numbers. To perform this algorithm only comparison, subtraction, bit shifting, and the ability to read the least significant bit is needed.

The algorithm relies on two simple facts. The first, is that the GCD of two odd numbers, cannot be even. The second, is that the difference of two odd numbers is both even, and divisible by the GCD of the original numbers.

Together, these form the core of the algorithm. The smaller of the two odd numbers, is subtracted from the larger, and then the result of the subtraction is bit shifted, until it is odd. The larger number is then discarded and the process is repeated until the two numbers are equal. At this point both the remaining numbers will equal the required GCD.

If either of the original numbers is even, then any powers of two are bit shifted away before the main loop commences. If this would result in a lost common factor of a power of two, then this is replaced at the end of the calculation, by bit shifting in the opposite direction.

As this algorithm directly exploits the binary representation of numbers, it is not going to be as efficient when using a modular representation. To bit shift a number is to either multiply or divide by a power of two, this can be achieved using the functions `scale` and **`unscale`**. Both of these functions take a machine integer as an argument, and modify the original number.

To inspect the last bit of a number, requires a reconstruction modulo 2. The function `modCRT_2_32` gives a reconstruction modulo 2^{32}. The function `modCRT_2_32` is extremely quick compared to `modCRT`, which can reconstruct to any small modulus. The time difference is around 10 times using the single processor Linux libraries, and around 40 times using the 32 bit Maspar library.

By reconstructing modulo 2^{32}, the last 32 bits are returned. This allows all the bit shifts needed, to be carried out in a single call to **`unscale`**. In the unlikely event that the last 32 bits are all zero, a second reconstruction can be performed, after **`unscale`** has been called.

7.2.3 Lehmer GCD

The Lehmer GCD algorithm is not as well known as the Euclidean, and binary algorithms. The reason for this, is that Lehmer's algorithm only works well for large numbers, whereas, the Euclidean and binary algorithms work well with numbers of a more normal size.

The idea behind the Lehmer algorithm is exactly the same as that used in the classical division algorithm, Algorithm 5. In the division algorithm, the most significant digits of the two numbers are used to predict the next digit of the quotient. In Lehmer's GCD algorithm, the most significant digits are used to calculate the steps that would have been taken in a Euclidean GCD algorithm.

If we are trying to find the GCD of X and Y, the method, is to first find two sets of numbers X_1,Y_1 and X_2,Y_2, which obey Equation 7.1.

$$\frac{X_1}{Y_1} < \frac{X}{Y} < \frac{X_2}{Y_2} \tag{7.1}$$

If $X = x_0 + x_1B + \ldots + x_nB^n$ and $Y = y_0 + y_1B + \ldots + y_nB^n$. Then valid values for the constants are, $X_1 = x_n$, $Y_1 = y_n + 1$, $X_2 = x_n + 1$, and $Y_2 = y_n$.

The next part of the method is to perform a standard Euclidean algorithm using the values X_1,Y_1 and X_2,Y_2. As long as the quotient at each step of the two

Euclidean loops is the same, then it must be the case that the original numbers X and Y, would have performed the same steps. If the quotients differ, then only an upper and lower bound on the true quotient is known. At this point the Euclidean loops are stopped.

To see this part of the algorithm in action, an example is taken from 'The Art of Computer Programming' [26]. In this example a number base of 10^4 is used.

Taking $X = 27182818$ and $Y = 10000000$ then $X_1 = 2718$, $Y_1 = 1001$, $X_2 = 2719$, and $Y_2 = 1000$. The result of the Euclidean loops can be seen in Table 7.1.

X_1	Y_1	Q_1	X_2	Y_2	Q_2
2718	1001	2	2719	1000	2
1001	716	1	1000	719	1
716	285	2	719	281	2
285	146	1	281	157	1
146	139	1	157	124	1
139	7	19	124	33	3

Table 7.1: Lehmer GCD calculation

The first 5 values of the quotient match, and so must be correct. The sixth quotient must lie between 3 and 19.

The 5 matching quotients, can be used to build up two equations for the new values of X and Y. These equations take the form of $X' = aX + bY$, and $Y' = cX + dY$, where X' and Y' are the new values of X and Y. The values of a, b, c and d, are calculated step by step, starting with the values $a = 1, b = 0, c = 0, d = 1$. In each step the equations $a' = c$, $b' = d$, $c' = a - Qc$, and $d' = b - Qd$ are used, where Q is the quotient calculated in the current step.

The new values of X and Y, can now be calculated using scaling and subtraction. This replaces several general divisions, saving significant time.

Lehmer GCD using the modular representation

The first problem in implementing Lehmer's algorithm when using a modular representation is obtaining the most significant bits of a number. The same problem was encountered in general division, where Equation 7.2 was used.

$$\frac{(approx_X - 1)M_X}{(approx_Y + 1)M_Y} < \frac{X}{Y} < \frac{(approx_X + 1)M_X}{(approx_Y - 1)M_Y} \tag{7.2}$$

If the lengths of the two numbers are equal, then Equation 7.2 simplifies to Equation 7.3.

$$\frac{approx_X - 1}{approx_Y + 1} < \frac{X}{Y} < \frac{approx_X + 1}{approx_Y - 1} \tag{7.3}$$

The values for X_1, Y_1, X_2 and Y_2 are then $X_1 = approx_X - 1$,$Y_1 = approx_Y + 1$,$X_2 = approx_X + 1$ and $Y_2 = approx_Y - 1$.

If the lengths differ by one, then the ratio of M_X/M_Y will be equal to $p_{len(X)-1}$. The values $X_1 = (approx_X - 1)p_{len(X)-1}$,$Y_1 = approx_Y + 1$,$X_2 = (approx_X + 1)p_{len(X)-1}$ and $Y_2 = approx_Y - 1$ could then be used. However, these values could be larger than the number base, so they would need to be bit shifted before the calculation takes place.

For speed of implementation, though not necessarily speed of calculation, when the number lengths differ by any amount, a standard Euclidean step is used. This is also necessary if none of the quotients match, during the dual Euclidean loops.

Once values have been found for X_1, Y_1, X_2 and Y_2, then the calculation can proceed in exactly the same way as it would using a traditional representation. When the values of a, b, c and d are known the functions **scale** and **sub** can be used to complete the calculation.

7.2.4 Weber GCD

Weber's GCD algorithm [43, 42] is a modified version of one the algorithms proposed by Sorenson in [36], it is similar to the binary algorithm in that it relies on bit shifting, however instead of producing numbers which have at least one trailing zero, instead numbers are produced which end with a larger number of zeros.

Like the binary algorithm, the main part of this algorithm assumes that the numbers are odd. Any powers of two are removed at the beginning of the algorithm, with any common powers of two being replaced at the end.

If the two numbers being reduced differ significantly in size, with $X > Y$,

then k is found such that Equation 7.4 holds.

$$k \equiv -X/Y \bmod 2^n, 0 \leq k < 2^n \tag{7.4}$$

If kY is subtracted from X, then the result of the calculation will be divisible by 2^n. This will allow at least n bits to be removed from the result.

If $X > kY$, then the $X - kY$, is both positive and less than X. If $X < kY$, then the result will be negative and less than kY. In the first case the resulting number is at least 2^n times smaller than X, in the second case it is at least $k/2^n$ times smaller than Y.

The best results are obtained when $X > 2^n Y$, as in this case it is certain that the value of X can be reduced by at least n bits per step. If X and Y are very similar in size then this method is no better than the binary algorithm shown above.

When X and Y are of similar size, then a different method is employed. This time a and b are found such that Equation 7.5 holds.

$$aX \equiv bY \bmod 2^m, 0 < a, |b| < \sqrt{2^m} \tag{7.5}$$

The value $|aX - bY|$ is divisible by 2^m, and as $0 \leq a, |b| < \sqrt{2^m}$, the value $|aX - bY|/2^m$ must be at least $\sqrt{2^m}/2$ times smaller than either X or Y.

For best results, setting the value of n to less than $(m-1)/2$ will allow the first method to be used after every application of the second method, this is desirable as the first method is both quicker and does not suffer from the problem of extra factors.

Extra factors

Ignoring, for the moment, how a and b are calculated, there is a problem with replacing the value of X with $|aX - bY|$. When X is replaced by $X - kY$, it is certain that $GCD(X, Y) = GCD(X - kY, Y)$, if this were not the case the Euclidean algorithm would not work.

But while any common divisor of X and Y will divide $aX - bY$, it is also possible to introduce extra factors, as $GCD(aX - bY, Y) = GCD(aX, Y)$.

At the end of the algorithm it is necessary to check that the value of the GCD calculated, divides the original values of X and Y. This extra step can be

speeded up by using the first method described, to reduce X to the same length as the supposed GCD, if the numbers are not equal then another GCD algorithm is used to produce a number which is both a multiple of $GCD(X, Y)$ and also divides X. The same process is repeated for Y, with a guarantee of the correct answer at the end.

Calculating a and b

To calculate a and b, a modified extended Euclidean algorithm is used. First the value of $X/Y \bmod 2^m$ is calculated, using standard methods. Then an extended Euclidean algorithm is performed, but instead of halting when $B = 1$ the algorithm terminates as soon as $B < \sqrt{2^m}$. If $X \equiv 187 \bmod 256$ and $Y \equiv 101 \bmod 256$, then $X/Y \equiv 159 \bmod 256$. Starting with the values $A = 256$, $B = 159$, $U = 0$ and $V = 1$, the result of modified extended Euclidean algorithm is shown in Table 7.2.

A	B	U	V
256	159	0	1
159	97	1	-1
97	62	-1	2
62	35	2	-3
35	27	-3	5
27	8	5	-8

Table 7.2: Modified inverse calculation

At each step, the invariant $VX \equiv BY \bmod 256$ holds. The last values of B and V are suitable for a and b, as their magnitudes are both less than $\sqrt{256} = 16$. That such numbers exist in every case, is proved in [41].

Implementation

In the implementation tested, the value of m was set to 2^{32}, and the value of n was set to 2^{16} for small differences in length, and 2^{32} for large differences. During the confirmation stage of the algorithm the binary GCD was used.

7.2.5 Testing

To test the four GCD algorithms, two numbers were generated which were slightly larger than a power of 2. Like in the other tests the numbers were built up by successive scaling. The two numbers were of the form p^aG and q^bG, where p and q were prime, and G was relatively prime to p and q.

Each of the four GCD algorithms works by reducing the size of its arguments in each step of the algorithm. To maximise the amount of work done a small GCD would be preferable.

If the GCD is too small then there is a chance that mistakes in the implementation of the algorithm will not be spotted. It may be possible that errors cancel out to give the correct result. This is especially true if the two numbers are relatively prime.

For these two reasons G was chosen to be 10000001, small enough to maximise the work done, and large enough to make the canceling out of errors improbable.

Single processor Linux

The results for the 32 bit single processor libraries are presented in Table 7.3.

Size(Bits)	Euclidean	Silver	Lehmer	Weber
32	2.4	0.5	1.5	1.6
64	20	2.1	2.1	2.1
128	57	6.6	5.6	4.5
256	168	24	15	14
512	503	79	35	43
1,024	1,782	287	121	153
2,048	6,357	1,198	451	544
4,096	23,472	4,510	1,780	1,986
8,192	93,420	18,641	7,120	8,720
16,384	372,026	75,383	26,232	35,335
32,768	1,500,311	305,113	111,418	148,139

Table 7.3: Time in milliseconds for Linux 32 bit Single GCD

Looking first at the results of the Euclidean algorithm, it can be seen that the time taken, grows quadratically with the input size. The overall time taken is very large with with the largest calculation taking 25 minutes to perform.

Even with the smallest numbers, Silver's algorithm is many times faster than the Euclidean algorithm. There is a ratio of around 5 between the times for all number sizes.

Lehmer's algorithm is slightly slower than Silver's for the smallest size input, but as the numbers grow the gap between the two, gets larger and larger with a difference approaching 3 times at the end.

Although slightly faster than Lehmer's algorithm below 256 bits, Weber's algorithm is slightly slower than Lehmer's for larger numbers. It is still twice as fast as Silver's algorithm for numbers above 2,048 bits.

Linux Cluster using AFAPI

The results for the 32 bit AFAPI libraries are presented in Table 7.4.

Size(Bits)	Euclidean	Silver	Lehmer	Weber
32	46	12	25	22
64	335	56	43	49
128	1,055	195	120	117
256	2,676	556	294	212
512	5,824	1,313	462	480
1,024	14,477	2,603	1,146	1,169
2,048	28,520	6,467	2,003	2,262
4,096	58,845	11,939	4,460	5,074
8,192	134,589	26,257	9,582	11,758
16,384	315,481	63,444	22,117	28,575
32,768	-	172,501	64,398	76,637

Table 7.4: Time in milliseconds for Linux 32 bit AFAPI GCD

Comparing the results on this table, with those in Table 7.4 the same comments apply. The Euclidean algorithm is significantly slower than the other three. The fastest algorithm for numbers above 512 bits is Lehmer's algorithm, with Weber's algorithm not far behind.

All four algorithms achieve parallel speedup at 16,384 bits. The three algorithms which have results for 32,768 bits all achieve a speedup of around 2.

Maspar MP-2

The results for the 16 bit Maspar MP-2 libraries are shown in Table 7.5.

Size(Bits)	Euclidean	Silver	Lehmer	Weber
32	116	37	54	43
64	835	125	150	130
128	2,094	314	376	256
256	4,840	764	588	553
512	10,258	1,453	1,382	1,054
1,024	24,214	3,027	2,707	2,167
2,048	47,184	5,980	5,496	4,406
4,096	91,042	11,992	10,887	8,687
8,192	188,179	24,281	22,242	17,332
16,384	379,164	48,519	43,922	34,765
32,768	757,129	96,844	103,680	69,879

Table 7.5: Time in milliseconds for Maspar 16 bit GCD

The difference between the Maspar results and those obtained using the Linux Cluster is the poor performance of Lehmer's algorithm. This is not unexpected, as Lehmer's algorithm performs a lot of sequential calculations, to avoid handling all the digits of the number. This is of no use on the Maspar as it takes the same time to handle all the digits, as it does for one.

Surprisingly Weber's algorithm, which also involves a lot of sequential calculations, is the best performing on the Maspar. This may be due to the primitive nature of the ACU, which is very slow at performing integer division. Lehmer's algorithm consists of many small Euclidean loops. Whereas the many inverses calculated in Weber's algorithm, are all inverses to a power of two, which can be performed using addition and bit operations.

PARI-GP

The same GCD calculations were also carried out using PARI-GP running on the same Linux workstation used to obtain the results in Table 7.3. Using the built in timer the results in Table 7.6 were obtained. It should be noted that the built in time only gave results to the nearest 10 ms.

Compared to the best of the single processor Linux results, PARI-GP is between 30 and 45 times faster. This is a significant difference in speed.

Some of the speed difference may be due to the efficiencies of the implementations. PARI-GP is partly written in assembler and has its own memory

Size(Bits)	gcd(x,y)
32	0
64	0
128	0
256	0
512	10
1,024	10
2,048	10
4,096	40
8,192	160
16,384	630
32,768	3,590

Table 7.6: Time in milliseconds for Linux PARI-GP GCD

management. The libraries described in this thesis are written entirely in C, and use standard memory management.

The rest of the difference is due to the modular representation. The GCD algorithms tested were all designed for a traditional representation and had to be modified before they could be used. This has decreased the efficiency of the original algorithms.

7.2.6 Comparison of GCD algorithms

The Euclidean algorithm is always the standard by which other algorithms will be measured, and it performed poorly. Some improvements could be made if a low level Euclidean GCD was written using custom division routines, but this is unlikely to make a large enough difference to be worthwhile.

Silver's algorithm would appear to be the most suitable for parallel operation as it does not involve much sequential calculation per step. It is hampered however, by only reducing the size of the numbers by a small amount in each step.

Lehmer's algorithm performed much better than the others on the Linux machines. With a lower processor count of four, the large amount of sequential calculation did not effect its overall efficiency. On the Maspar however, it performed worse than both Weber's and Silver's algorithms.

Weber's algorithm performed better on the Maspar than it did on the Linux

machines. Overall it is comparable with Lehmer's algorithm when using a modular representation.

Overall when using a modular representation the choice of GCD algorithm will depend on the platform, for the Maspar Weber's is best, and for the Linux machines, Lehmer's is best. The size of the arguments has little effect on the relative performance of the algorithms removing any benefit of a hybrid approach.

When compared to PARI-GP all the algorithms performed poorly. The poor performance of the Euclidean algorithm is due to the high cost of general division. Of the other algorithms only Lehmer's could be directly implemented using a modular representation which explains its relatively good performance.

7.3 Calculating Determinants

7.3.1 Introduction

Calculating determinants is a classic use of modular arithmetic. With no comparisons or general divisions needed, the value of the determinant can be quickly calculated for each residue independently.

In this section, two approaches are compared. In the first approach, the standard datatype is used, this will allow the length of the sub products to be automatically managed by the library. In the second approach, a bound on the size of the resulting determinant is first calculated, and then the number of residues is fixed for the duration of the determinant calculation.

7.3.2 How the determinants are calculated

To calculate the determinant, each element of the top row of the n by n matrix is multiplied by the determinant of the appropriate $n-1$ by $n-1$ sub matrix. These products are then either added or subtracted from a sum depending on the position of the element.

This process is applied recursively until the matrix is 1 by 1, when no calculation is needed to determine the determinant. It is assumed that the matrix is dense.

7.3.3 Using a fixed number of residues

As mentioned above two implementations were tested. The first used the standard datatype and the second used the fixed number of residues datatype. To be able to use the fixed number of residue datatype a bound is needed on the size of the determinant.

A maximum value of the absolute size of a determinant is given by Hadamard's bound, which is shown in Equation 7.6.

$$|det(X)| \leq \prod_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} |x_{ij}|^2 \right)^{1/2} \tag{7.6}$$

It is not possible to use the bound as it is written, as the libraries do not have a square root function. Rather than adding such a function to the libraries, a simplified bound is used. This is shown in Equation 7.7.

$$|det(X)| \leq \prod_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} |x_{ij}| \right) \tag{7.7}$$

As $\left(\sum_{j=0}^{n-1} |x_{ij}|^2\right)^{1/2} < \left(\sum_{j=0}^{n-1} |x_{ij}|\right)$, this simplified bound, will always be larger than Hadamard's bound.

7.3.4 Testing

To test the determinant code, a 6 by 6 matrix was filled with random integers between 0 and 9999. To ensure that the numbers were the same every time, the random number generator was seeded with the value 0.

A determinant was then calculated using these small values and outputted to the screen. To ensure that the value was correct, the whole matrix was printed out and inputted to Maple.

To increase the size of the values, without altering the determinant, rows were added to each other at random. The number of times this was performed determined the final size of the elements.

In the standard case, the determinant of these large values was calculated and the result displayed on the screen. It was then easy to check the validity of this determinant as it should match the value calculated using the small values.

For the fixed number of residues code, a loose Hadamard bound was calculated, using the numbers stored in the matrix. This bound was then used to set the number of residues needed for the calculation, the whole matrix being converted from `t_PMA` to `t_LPMA`.

The determinant was then calculated, and the result converted back to `t_PMA` and printed to the screen. The entire test program can be seen in Appendix B.4.2.

7.3.5 Results

The size of the matrix was set at 6 by 6, although the test code was capable of using a matrix of any size. To test the 16 bit libraries, 10000 additions were used. This produced elements with lengths of around 140 moduli, which is approximately 2,200 bits. For the 32 bit code, 10000, and 20000 additions were used, with 20000 additions producing elements of around 140 32 bit moduli, or 4,400 bits.

The results for the Linux Cluster are shown in Table 7.7. The first column of results, is the time taken using the standard datatype. The five rightmost columns, represent the time taken, for each stage of the fixed number of residues code. The time taken to calculate the bound is in the first column, then the time taken converting the numbers to type `t_LPMA`. The third value is the time taken to calculate the determinant, and the fourth is the time taken to convert the number back to type `t_PMA` and display it on the screen.

Library	Size	Standard	Fixed				
		Deter	Bound	Convert	Deter	Output	Total
16-Single	10,000	31,623	170	720	1,292	90	2,272
16-AFAPI	10,000	37,837	128	358	370	50	908
32-Single	10,000	36,455	342	1,704	1,122	70	3,238
	20,000	131,833	1,407	6,896	2,217	130	10,650
32-AFAPI	10,000	42,150	210	597	253	45	1,105
	20,000	93,966	470	2,214	500	70	3,254

Table 7.7: Time in milliseconds for Determinant Calculations

Calculating using the fixed number of residues determinant, is clearly much faster than using the standard code. The difference between the two columns labelled Deter is the difference in time taken performing the same calculation

using the two different datatypes. With a single processor the difference is around 30 times using 10,000 additions and around 60 times using 20,000 additions.

Using the standard code the AFAPI libraries manage to achieve speedup when 20000 additions are used, but fail to do so when 10000 additions are used. This is consistent with previous results.

The parallel speedup achieved during the fixed number of residues determinant, is slightly greater than 4. The reason that it is so good, is that no communication is required during the entire determinant calculation.

The majority of the time taken in performing the fixed number of residue calculation is in converting to type `t_LPMA`. This is due to the base extensions needed to increase the number of residues to the bound set. As these base extension use relatively small numbers, this reduces the overall speedup achieved by the AFAPI code to around 3 times.

Despite the large numbers of primes used to calculate the determinant, converting the result back to standard form is relatively quick. This is in part due to the probabilistic algorithm used to determine the number's length.

7.3.6 Conclusion

In this section an example of the use of the fixed number of residues datatype has been seen. Despite using a very loose bound, significant advantage was seen both in terms of the sequential time of calculation, and also in parallel efficiency.

7.4 Conclusions

Two examples have been presented in this chapter, calculating GCDs, and calculating determinants. Both of these are common operations which, are likely to be required during many calculations.

With the GCD algorithms is was shown that standard algorithms intended for a traditional number representation can be coded directly using the libraries. While parts of Lehmer's algorithm required access to the underlying datatype, none of the other functions required any low level knowledge, to be successfully employed.

When calculating determinants, the full power of the modular representation

can be used. In these cases it is possible to fix the number of residues and perform the calculation without the need for communication. This allowed 100% parallel efficiency to be achieved using number sizes which are usually too small for any parallel speedup on the Linux Cluster.

Caution must be taken however, when only a small part of a calculation can be performed in this way. It may be the case that performing the calculation using the standard datatype takes less time than converting the numbers between the two datatypes.

Part III

Conclusions

Chapter 8

Conclusions

8.1 Aims and objectives

The aim of this thesis was to write a library of functions for performing bignum calculations, which was to run on a parallel computer. Of particular interest was the implementation of division and related functions, which are not naturally suited to parallel implementation.

One approach to this problem was to use FFT multiplication, coupled with a Newtonian inverse to perform division. This was the subject of the thesis of G. Cesari [5]. Newtonian inverse based division, is only effective when using numbers of significant size, Cesari showed that a standard division algorithm was more effective when the numbers had less than 32,768 bits.

As it is not efficient to perform a standard division in parallel, one of the objectives of this thesis was to create a parallel division algorithm, which achieved parallel speedup with numbers of less than 32,768 bits.

A modular representation was chosen as it allows each residue to be dealt with in isolation. Several algorithms had been proposed for circuits to be built, which could perform division and RSA style calculations. These circuits each assumed the number of residues to be fixed. Another objective of this thesis was to cope with numbers which use different numbers of residues, so calculations can be performed in which the size of the result is not known.

Two further objectives are that the library should be able to use a large number of processors, and that the library should work on a large range of platforms.

8.2 New Results

As the purpose of research is to extend the boundaries of knowledge, in this section mention will be made of some of the new results which are to be found within this thesis.

There are many libraries for bignum arithmetic, many of which are part of larger symbolic calculation packages, such as Reduce or Axiom. However, this is the first library to allow general calculations to be performed using a modular representation.

It is also one of a handful of packages which allow bignum calculations to be performed on a parallel computer.

While the coupling of a modular representation together with an approximation of its size is a reasonably obvious idea, this is the first time that it has been implemented. Other approximate reconstructions have used a floating point estimate, or pre-calculated tables. The method of calculating the integer approximations, has not been previously published.

Reconstructing a number to a small modulus is a new method of high speed reconstruction, it is used extensively throughout the libraries and is a core part of the division algorithm.

If the number of residues is not to be fixed, then the lengths of the numbers have to be monitored. Without knowing the lengths of the numbers it would not be possible to known whether the number of residues was sufficient to store the result.

The concept of length was also discussed by Bronnimann in [3]. As mentioned in the introduction, this paper presented several length algorithms which were also published concurrently by myself in [33]. What was not published was the probabilistic length algorithm presented in Section 2.3, which had been edited out of [33] to reduce the paper's length.

For most calculations it is relatively simple to predict the length of the result, for multiplication, however, larger problems are encountered. This is the first time this problem has been tackled.

The probabilistic divisibility test is specific to the representation, but it shows advantages over deterministic tests using any representation.

Finally, the known division function, has only been proposed as part of a

circuit design, and as such, this is the first implementation of the algorithm.

8.3 Discussion of results

8.3.1 Platforms

The libraries were originally intended to be used on the Maspar MP-1. The age and vulnerability of this machine led to a search for alternative platforms. The Maspar is an ideal platform for parallel arithmetic because it is capable of extremely rapid communication. The machine is however very old and the 4 bit processors are individually very weak.

The cluster of Linux workstations are used individually for everyday work, but are rarely used either remotely or at night. This made them an ideal platform for obtaining consistent results. It was the move to this platform which highlighted the need for a variable number of residues.

Communicating through Ethernet/MPI is not feasible for parallel arithmetic, as communication times start at several milliseconds. The results show this to be the case, with parallel speedup only being achieved using the largest of numbers.

To decrease the communication time the TTL_PAPERS circuit board was built, this gives superior communication times for messages of the size used in the library. This made it possible to achieve much better results, but still well short of optimal.

As was shown in the results, trying to use all the processors on a shared memory machine, is not feasible in a multi user environment. When using 4 out of 20 processors, much better results were achieved. To get the results shown took several months, as the machine was very heavily used. The average load was between 25 and 30. Only when the load dipped below 16, were the tests run, with a load of 15 needed for consistent results. It was not possible to get results using more processors as the load never fell low enough.

It is a great shame that more results could not be obtained using the 20 processor machine as it gave by far the best results. In the establish length tests speedup over a single processor was achieved at 512 bits(64 bytes), with 95% efficiency at 16,384 bits(2,048 bytes). In the base extension tests >100% efficiency was achieved when extending from 65 to 256, 32-bit residues. This

super-linear speedup is put down to only having to cache a quarter of the tables per processor. With such high efficiencies being obtained with 4 processors, it is clear that libraries would scale to more processors.

Chronologically, the last platform to be used was the Maspar MP-2, this had 4 times as many processors as the MP-1. It gave very similar results, being between 2 and 4 times faster than the MP-1, with the added capability of 4 times the number range.

8.3.2 Comparison of MPI and AFAPI

On the Linux cluster, the different results for MPI and AFAPI were due to the communication mechanisms used. The MPI libraries communicated using standard Ethernet, whereas the AFAPI libraries used a custom circuit board.

It was seen that the circuit board was significantly faster at broadcasting messages of the size used in the library. When performing an approximate reconstruction with 16 bit moduli, it was seen that the fixed communication costs when using MPI were over 6 times larger than those when using AFAPI.

On the shared memory machines the communication mechanism is the same for both MPI and AFAPI. The communication costs were again much higher using MPI.

On the Solaris server, the fixed communication costs when performing an approximate reconstruction with 16 bit moduli, was 10 microseconds for AFAPI, and 600 microseconds for MPI. With 32 bit moduli the costs doubled for AFAPI but stayed the same for MPI. Even with 32 bit moduli the MPI communication costs are 30 times those for AFAPI.

On the Irix server the communication costs for the approximate reconstruction with 16 bit moduli, was 30 microseconds for AFAPI, and 250 microseconds for MPI. When 32 bit moduli were used the communication costs were almost identical. This gives a ratio of 8 times.

The difference in communication costs using MPI and AFAPI on shared memory computers is down to the design of the libraries. MPI is based around message passing, whereas AFAPI is based around synchronisation and broadcasting.

It has been seen that for the style of modular arithmetic presented in this thesis that the AFAPI design is more effective.

8.3.3 Test Results

The most consistent aspect of the test results was the close correlation between the timings of the three basic functions, and the time taken performing the higher order functions.

One of the objectives set, was to achieve parallel speedup using a division algorithm, with arguments less than 32,768 bits. Looking at the results for the Linux Cluster, it can be seen that parallel speedup is obtained using the 16 bit libraries at 8,192 bits and with the 32 bit libraries at 16,384 bits. Using the 32 bit libraries, a 2 times speedup is obtained at 32,768 bits, and a projected 3 times speedup would be achieved at 65,536 bits.

These results are very similar to the results obtained for establish length, which achieved parallel speedup at 4,096 bits with the 16 bit libraries and at 8,192 bits with the 32 bit libraries. Numbers of size 32,768 bits giving a 2.3 times speedup and a projected speedup of 3 times at 65,536 bits.

If this correlation was to be carried over to the 20 processor shared memory machine, speedup could be expected at around 512 to 1024 bits. With efficiencies of well over 95% at 32,768 bits.

The efficiency of the Maspar implementation is hard to assess. At 32,768 bits it is faster than the single processor Linux library, and at 65,536 bits it is faster than the Linux AFAPI results. However, with 4096 processors it might be expected to do better than this. In general the Maspar only seems to do well when almost all its processors are being used.

This general lack of performance has to be put down to a fundamental problem with division. The Maspar's processors are very simple and are not built to perform fast integer division. In a library that uses a lot of modular reductions this is a huge disadvantage.

What the Maspar does show is the potential parallelism of the libraries, linear sequential problems take constant time on the Maspar, and quadratic sequential problems take linear time. The communication costs on the Maspar are fixed with no time saving possible when only a few processors are used.

When the results of the GCD calculations are compared, it is not possible to say that one algorithm is faster than all the others. When using the AFAPI libraries on the Linux cluster, Lehmer's algorithm is the fastest. When using the Maspar MP-2, Weber's algorithm is the fastest. For this reason, several GCD

algorithms will need to be included in the library. Each of the algorithms can then be tested on the specific hardware being used before a final choice is made.

8.4 Further Work

8.4.1 Controlling the number of residues used to store a number

One of the problems encountered when writing algorithms with the library, was controlling the amount of residues used to store a number. The arithmetic functions return a number with the smallest possible number of residues. This is the most convenient result when numbers are shrinking, such as during GCD calculations.

In other situations this behaviour is not useful. In division, an estimate is made of the quotient during each step. Great care had to be taken for this number to be calculated using enough residues to store the product of the quotient and the divisor. If this is not done, the quotient would need to be base extended during each step, increasing the time complexity of division to cubic!

The functions `scale` and `unscale`, take an integer as input and modify an already existing number. The number of residues used to store the number, will only be increased, never decreased. Similar functions could be written which take two numbers of type `t_PMA` as arguments. This would allow the user to both avoid reallocating memory, and to preserve the number of residues.

8.4.2 Reducing the size of tables

One of the reasons the Maspar performed poorly was that it never used all of its processors. Even the largest test, used only just over half of them. If the number of residues could have been increased to 8,192 or 16,384 then full efficiency could have been achieved throughout a calculation.

The reason such numbers could not be stored on the Maspar, was a lack of memory. This restriction also affected the Linux Cluster as they only have 32 MB of memory each. If the size of the tables could be reduced, much larger numbers could be handled.

One solution to reduce the size of the tables, is not to store all the values. If only every fourth reconstruction constant was stored, then a maximum of 3 extra calculations would be needed to calculate any constant required. However, each time the number of residues is doubled, the gaps between the constants stored in the table increases four times. The amount of extra calculation this involves would rapidly become overwhelming.

Another solution might be to cache the reconstruction constants as they are calculated. This should mean that only the values used are stored. If the numbers used are around the same size for the whole calculation, then it may be possible to extend the number range many times.

Other, more radical solutions, would involve changing the algorithms to only perform reconstructions using certain numbers of residues. To be able to calculate length for instance, it may be possible to increase the size of the numbers, rather than decreasing the number of residues. This would involve multiplying the number being approximated by certain fixed amounts, such as powers of two. How these fixed amounts are calculated then becomes a problem.

When performing a base extension, it is possible to reconstruct the number using all of the calculated residues. This would again reduce the amount of reconstruction constants needed, as all numbers are stored with a number of residues of the form $2^n nproc$.

8.4.3 Polynomial arithmetic

A natural extension of the library described in this thesis would be to build a library of functions for polynomial arithmetic. While the library is very efficient when performing addition and multiplication, it is not as fast at performing general division. In polynomial arithmetic no general division is needed as the pseudo-division used for polynomials uses only exact divisions.

The standard library coupled with a GCD algorithm from Chapter 7 will allow the majority of polynomial algorithms to be implemented.

8.5 Summary

The aim of this thesis was to create a library of functions to perform bignum calculations on a parallel computer. The modular representation was chosen as it appeared the most suitable for a parallel implementation.

By starting with an *Approximate CRT Reconstruction*, the tools have been built up to allow an arithmetic library to be written. The length of a number stored in a modular representation was defined. Also defined, was a datatype containing both the number stored in a modular representation, and its length. Using this datatype, algorithms could be written which were similar to the algorithm used when numbers are stored in a traditional representation.

All of the arithmetic operations were built using three main algorithms. These algorithms were used, to approximate the size of a number, extend the number of residues, and to perform reconstructions to small moduli.

The library was tested, when performing a range of arithmetic tasks including testing for divisibility, and division by a known divisor.

Example programs were written which performed GCD calculations, these demonstrated how the libraries could be used. In another program to calculate determinants, a simplified datatype was used to increase both parallel and sequential efficiency.

Appendix A

Design of TTL_PAPERS circuit board

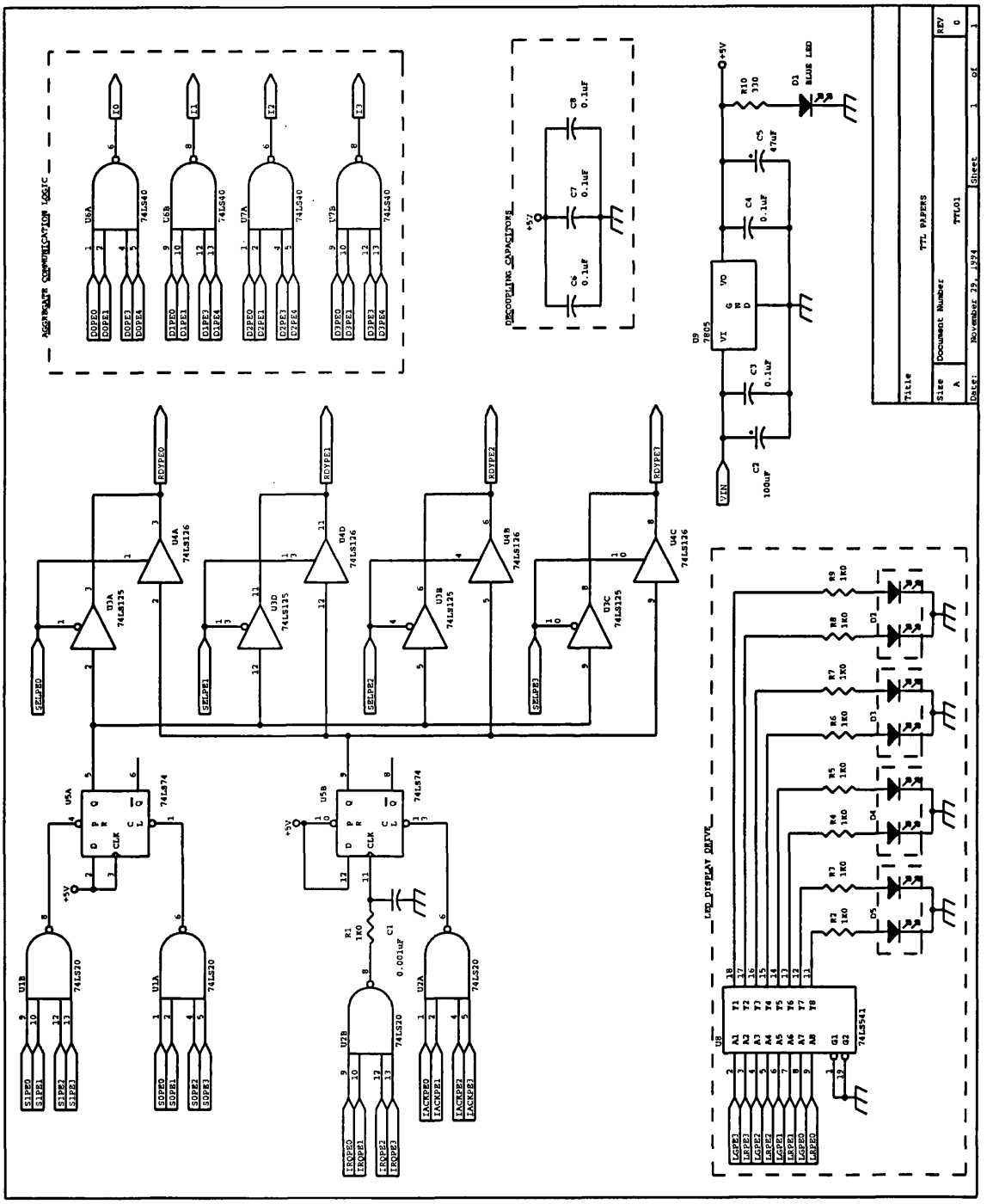

Figure A-1: TTL_PAPERS circuit diagram

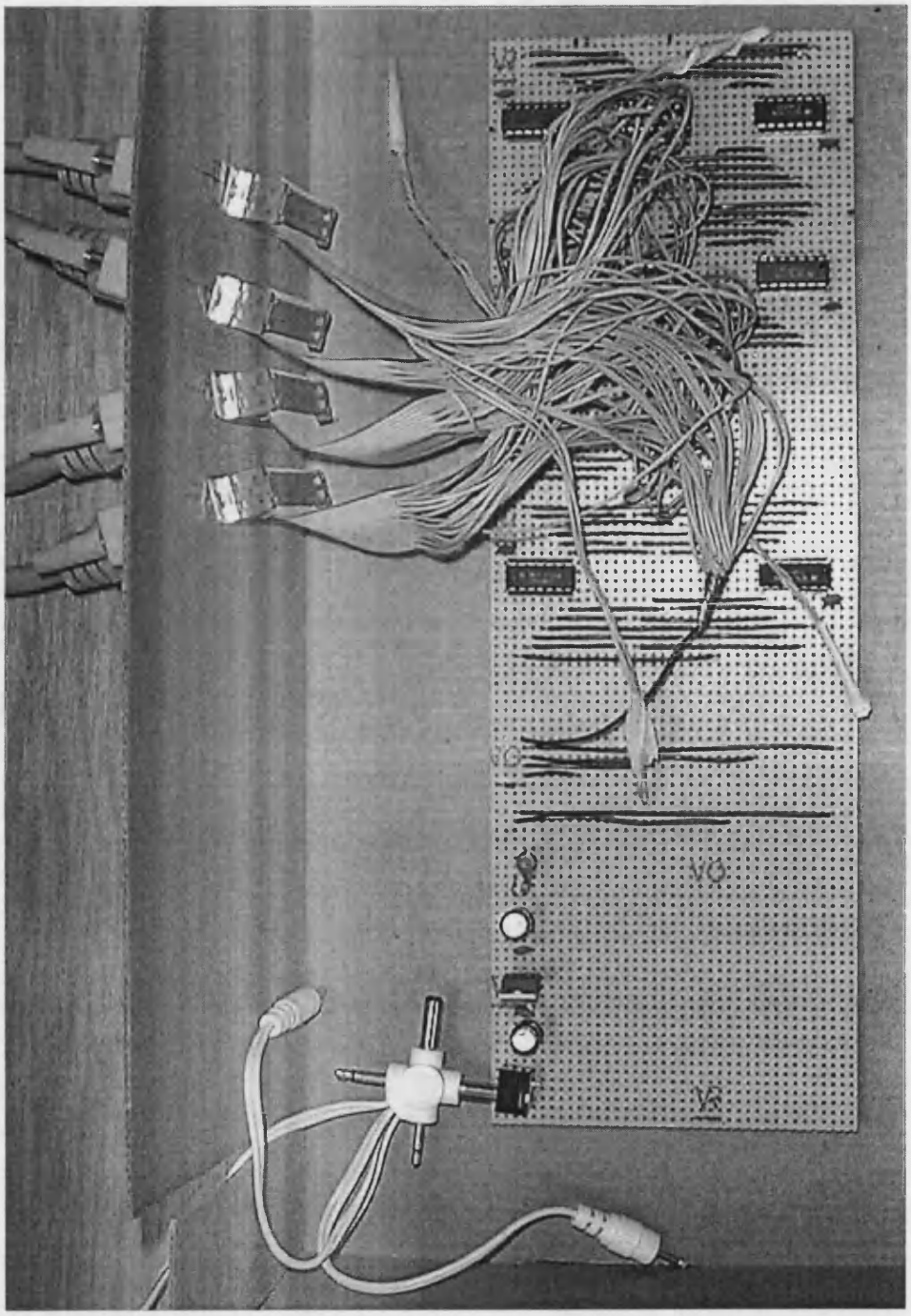

Figure A-2: TTL_PAPERS circuit board

Appendix B

The PMA library

B.1 Overview

B.2 t_PMA functions

```
/* Main Datatype */
typedef struct T_PMA {
  int           neg;
  unsigned int *array;
  int           num_res;
  int           length;
  unsigned int  approx;
  int           k;
} *t_PMA;

/* First/Last part of each program */
int initialise(int*,char***);
int finalise(void);

/* Creation functions */
t_PMA create(void);
t_PMA duplicate(t_PMA);
```

```
void  destroy(t_PMA);

/* I/O functions */
void  input(t_PMA);
void  residues(t_PMA);
void  output(t_PMA);
int   is_zero(t_PMA);
void  info(t_PMA);
#define cprintf      printf
#define getint(ret)  scanf(" %d",ret)

/* Used to increase/decrease number of residues */
void base_reduction(t_PMA,int);
void base_extension(t_PMA,int);

/* Arithmetic functions */
int compare(t_PMA,t_PMA);
#define add(A,B,C) addsub(A,B,C,0)
#define sub(A,B,C) addsub(A,B,C,1)
void assign(t_PMA,t_PMA);
void negate(t_PMA);
void assign_32u(t_PMA,unsigned int);
void addsub(t_PMA,t_PMA,t_PMA,int);
void mult(t_PMA,t_PMA,t_PMA);
void div_exact(t_PMA,t_PMA,t_PMA);
void scale(t_PMA,unsigned int);
void r_div(t_PMA,t_PMA,t_PMA);
void q_div(t_PMA,t_PMA,t_PMA);
void qr_div(t_PMA,t_PMA,t_PMA,t_PMA);
unsigned int modCRT(t_PMA,unsigned int);
unsigned int modCRT_32u(t_PMA,unsigned int);
unsigned int mod_2_32(t_PMA);
void gcd(t_PMA,t_PMA,t_PMA);
void un_scale(t_PMA,unsigned int);
```

```
#define iszero(A)       (A->approx == 0)

/* Timer Functions */
void timer_start(void);
void timer_stop(void);
void timer_reset(void);
struct timeval timer_t_return(void);
struct timeval timer_u_return(void);
struct timeval timer_s_return(void);
void timer_print(void);
```

B.3 t_LPMA functions

```
/* Main datatype */
typedef struct T_LPMA {
  unsigned int *array;
  int           num_res;
} *t_LPMA;

/* Creation functions */
t_LPMA L_create(int);
t_LPMA L_duplicate(t_LPMA);
void L_destroy(t_LPMA);

/* Conversion functions */
int L_get_bound(t_PMA);
void L_convert_to(t_LPMA,t_PMA);
void L_convert_from(t_PMA,t_LPMA);

/* I/O functions */
void L_input(t_LPMA);
void L_residues(t_LPMA);
```

```
void L_output(t_LPMA);
void L_info(t_LPMA);

/* Arithmetic functions */
int L_is_zero(t_LPMA);
void L_assign(t_LPMA,t_LPMA);
void L_assign_32u(t_LPMA,unsigned int);
int L_is_equal(t_LPMA,t_LPMA);
void L_negate(t_LPMA);
void L_add(t_LPMA,t_LPMA,t_LPMA);
void L_sub(t_LPMA,t_LPMA,t_LPMA);
void L_mult(t_LPMA,t_LPMA,t_LPMA);
void L_div_exact(t_LPMA,t_LPMA,t_LPMA);
void L_scale(t_LPMA,unsigned int);
void L_unscale(t_LPMA,unsigned int);
```

B.4 Examples

B.4.1 Euclidean GCD

```
#include<PMA.h>

int main(int argc,char **argv) {

  t_PMA A,B,q,r;
  int i;

  if(initialise(&argc,&argv)) {
    fprintf(stderr,"PMA initialisation error\n");
    exit(1);
  }

  A = create();
```

```
  B = create();
  q = create();
  r = create();

  cprintf("Please enter A\n");
  input(A);
  cprintf("Please enter B\n");
  input(B);

  timer_start();
  euclidean(q,A,B);
  timer_stop();
  cprintf("Time to perform gcd\n");
  timer_print();
  timer_reset();

  cprintf("GCD is\n");
  output(q);

  if(finalise()) {
    fprintf(stderr,"PMA finalisation error\n");
    exit(1);
  }

}

void euclidean(t_PMA out,t_PMA in1,t_PMA in2) {

  t_PMA t,A,B;

  A = duplicate(in1);
  B = duplicate(in2);

  /* Make sure A >= B */
```

```
  if(compare(A,B) < 0) {
    t = A;
    A = B;
    B = t;
  }

  while(!(iszero(B))) {
    r_div(A,A,B);
    t = A;
    A = B;
    B = t;
  }
  assign(out,A);

  destroy(A);
  destroy(B);
}
```

B.4.2 A prime locked determinant calculation

```
#include<PMA.h>

void print_matrix(t_PMA**,int);
void addcolumns(t_PMA**,int);
void determinant(t_LPMA,t_LPMA**,int,int,int*);
unsigned int BOUND;

int main(int argc,char **argv) {

  int size,i,j,num_add,*missing;
  t_PMA **Matrix,determ;
```

```
t_LPMA **MatrixL,determL;

if(initialise(&argc,&argv)) {
  fprintf(stderr,"PMA initialisation error\n");
  exit(1);
}

cprintf("Please enter size of matrix\n");
getint(&size);
cprintf("Initialising %dx%d matrix\n",size,size);
srand(0);

Matrix = (t_PMA**) malloc(size*sizeof(t_PMA*));
for(i=0;i<size;i++) {
  Matrix[i] = (t_PMA*) malloc(size*sizeof(t_PMA));
  for(j=0;j<size;j++) {
    Matrix[i][j] = create();
    assign_32u(Matrix[i][j],rand()%10000);
  }
}

cprintf("Please enter number of additions\n");
getint(&num_add);

for(i=0;i<num_add;i++)
  addcolumns(Matrix,size);
print_matrix(Matrix,size);

timer_reset();
timer_start();
/* Going to use a simplified Hadamand bound */
{
  t_PMA sum,prod,value;
```

```
  sum =    create();
  prod =   create();

  assign_32u(prod,1);
  for(i=0;i<size;i++) {
    assign_32u(sum,0);
    for(j=0;j<size;j++) {
      add(sum,sum,Matrix[i][j]); /* All values are +ve */
    }
    mult(prod,prod,sum);
  }
  BOUND = L_get_bound(prod);
  destroy(sum);
  destroy(prod);
}
timer_stop();
cprintf("BOUND is %d\n",BOUND);
cprintf("Time to calculate (loose) Hadamard bound\n");
timer_print();
timer_reset();
timer_start();

/* Create LOCKED MATRIX */
MatrixL = (t_LPMA**) malloc(size*sizeof(t_LPMA*));
for(i=0;i<size;i++) {
  MatrixL[i] = (t_LPMA*) malloc(size*sizeof(t_LPMA));
  for(j=0;j<size;j++) {
    MatrixL[i][j] = L_create(BOUND);
    L_convert_to(MatrixL[i][j],Matrix[i][j]);
  }
}
timer_stop();
cprintf("Time to convert to t_LPMA\n");
timer_print();
```

```
  timer_reset();

  /* Calc determinant */
  missing = (int*) malloc(sizeof(int)*size);
  determL = L_create(BOUND);

  timer_start();
  determinant(determL,MatrixL,size,0,missing);
  timer_stop();
  cprintf("Time to calculate determinant\n");
  timer_print();
  timer_reset();

  timer_start();
  L_output(determL);
  timer_stop();
  cprintf("Time to print determinant\n");
  timer_print();
  timer_reset();

  if(finalise()) {
    fprintf(stderr,"PMA finalisation error\n");
    exit(1);
  }

}

void print_matrix(t_PMA **Matrix,int size) {

  int i,j;

  cprintf("[\n");
  for(i=0;i<size;i++) {
    cprintf("[\n");
```

```
    for(j=0;j<size;j++) {
      output(Matrix[i][j]);
      cprintf("\n");
    }
    cprintf("]\n");
  }
  cprintf("]\n");
}

void addcolumns(t_PMA **Matrix,int size) {

  int column1,column2;
  int i;

  column1 = rand()%size;
  column2 = rand()%size;
  while(column1 == column2)
    column2 = rand()%size;

  for(i=0;i<size;i++)
    add(Matrix[column1][i],Matrix[column1][i],Matrix[column2][i]);
}

void determinant(t_LPMA result,t_LPMA **Matrix,int size,
                 int level,int *missing) {

  int y_pos=0,i,j,k,sign=1;
  t_LPMA sub;

  if(size == level + 1) {
    for(j=0;j<level;j++)
      for(i=0;i<level;i++)
        if(missing[i] == y_pos)
          y_pos++;
```

```
    L_assign(result,Matrix[level][y_pos]);
  }
  else {
    sub = L_create(BOUND);
    L_assign_32u(result,0);
    for(k=0;k<size-level;k++) {
      for(j=0;j<level;j++)
        for(i=0;i<level;i++)
          if(missing[i] == y_pos)
            y_pos++;

      missing[level] = y_pos;
      determinant(sub,Matrix,size,level+1,missing);
      L_mult(sub,sub,Matrix[level][y_pos]);
      if(sign == -1)
        L_negate(sub);
      L_add(result,result,sub);
      sign *= -1;
      y_pos++;
    }
    L_destroy(sub);
  }
}
```

Bibliography

[1] G. Alia and E. Martinelli. Sign detection in residue arithmetic circuits. *Journal of System Architecture*, 45:251–258, 1998.

[2] P.W. Beame, S.A. Cook, and H.J. Hoover. Log depth circuits for division and related problems. *SIAM J Comput*, 15(4):994–1003, Nov 1986.

[3] H. Bronnimann, I.Z. Emiris, V.Y. Pan, and S. Pion. Sign determination in residue number systems. *Theoretical Computer Science*, 210:173–197, 1999.

[4] D.B. Burton. *Elementary Number Theory.* Wm. C. Brown, 1994.

[5] G. Cesari. *Parallel Algorithms for Multiple-Precision Arithmetic.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, 1997.

[6] G. Cesari and R. Maeder. Performance analysis of the parallel karatsuba multiplication algorithm for distributed memory architectures. *Journal of Symbolic Computation*, 21:467–473, 1996.

[7] B. Char, J. Johnson, B.D. Saunders, and A. Wack. Some experiments with parallel bignum arithmetic. In *PASCO '94*, pages 94–103, 1994.

[8] G.I. Davida and B. Litow. Fast parallel arithmetic via modular representation. *SIAM J. Comput*, 20(4):756–765, Aug 1991.

[9] H.G. Dietz, T.M. Chung, T. Mattox, and T. Muhammad. Purdue adapter for parallel execution and rapid synchronisation: The TTL_PAPERS design. Technical report, School of Electrical Engineering, Purdue University, 1995.

[10] H.G. Dietz, T. Mattox, and G. Krishnamurthy. The aggregate function API: It's not just for PAPERS anymore. Technical report, School of Electrical Engineering, Purdue University, 1997.

[11] H.G. Dietz, T. Muhammad, and T. Mattox. TTL implementation of purdue's adapter for parallel execution and rapid synchronisation. Technical report, School of Electrical Engineering, Purdue University, 1994.

[12] G. Dimauro, S. Impedovo, and G. Pirlo. A new technique for fast number comparison in the residue number system. *IEEE Transactions on Computers*, 42(5):608–612, 1993.

[13] V.S. Dimitrov, G.A. Jullien, and W.C. Miller. A fast and robust rns algorithm for evaluating signs of determinants. *Computers Mathematics and Applications*, 35(8):9–14, 1998.

[14] S. Even. Systolic modular multiplication. *Lecture Notes in Computer Science*, 537:619–624, 1991.

[15] M. J. Flynn. Very high-speed computing systems. *Proc. IEEE*, 54:1901–1909, Dec 1966.

[16] T. Granlung. *The Gnu Multiple Precision Arithmetic Library.* TMG Datakonsult, 2.0.2 edition, June 1996.

[17] G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers.* Oxford Science Publications, 5th edition, 1979.

[18] A.A. Hiasat and H. Abdel-Aty-Zohdy. Semi-custom vlsi design and implementation of a new efficient rns division algorithm. *The Computer Journal*, 42(3):232–240, 1999.

[19] M.A. Hitz and E. Kaltofen. Integer division in residue number systems. *IEEE Transactions on Computers*, 44(8):983–989, 1995.

[20] C.Y. Hung and B. Parhami. An approximate sign detection method for residue numbers and its application to RNS division. *Computers Math. Applic.*, 27(4):22–35, 1994.

[21] C.Y. Hung and B. Parhami. Error analysis of approximate chinese-remainder theorem decoding. *IEEE Transactions on Computers*, 44(11):1344–1348, 1994.

[22] C.Y. Hung and B. Parhami. Fast rns division algorithms for fixed divisors with application to rsa encryption. *Information Processing Letters*, 51:163–169, 1994.

[23] K. Hwang. *Computer Arithmetic, Principles, Architecture and Design.* John Wiley and Sons, 1979.

[24] T. Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15:169–180, 1992.

[25] T. Jebelean. A generalization of the binary GCD algorithm. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1993.

[26] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 3rd edition, 1998.

[27] W. Krandick and T. Jebelean. Bidirectional exact integer division. *Journal of Symbolic Computation*, 21:441–455, 1996.

[28] Maspar Computer Corporation. *Maspar System Overview*, 9300-0100-a6 edition, November 1992.

[29] Maspar Computer Corporation. *Maspar Parallel Application Language (MPL), User Guide*, 9302-0101-a5 edition, July 1993.

[30] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[31] K.C. Posch and R. Posch. Base extension using a convolution sum in residue number systems. *Computing*, 50:93–104, 1993.

[32] K.C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, 1995.

[33] D. Power and R. Bradford. A library for parallel modular arithmetic. In *Lecture Notes in Computer Science 1685, EuroPar'99*, pages 1476–1483, 1999.

[34] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key systems. *Communications of the ACM*, 21:120–126, 1978.

[35] J. Schwemmlein, K.C. Posch, and R. Posch. Rns-modulo reduction upon a restricted base value set and its applicability to rsa cryptography. *Computers and Security*, 17(7):637–650, 1998.

[36] J. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16:110–144, 1994.

[37] N.S. Szabo and R.I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* McGraw Hill, New York, 1967.

[38] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[39] T.V. Vu. Efficient implementations of the chinese remainder theorem for sign detection and residue decoding. *IEEE Transactions on Computers*, C-34(7):646–651, 1985.

[40] C.D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3), 1993.

[41] P.S. Wang, M.J.T. Guy, and J.H. Davenport. P-adic reconstruction of rational numbers. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 1982.

[42] K. Weber. The accelerated integer GCD algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995.

[43] K. Weber. Parallel implementation of the accelerated integer GCD algorithm. *Journal of Symbolic Computation*, 21:457–466, 1996.